uc3m | Universidad Carlos III de Madrid

Bachelor's Degree in Computer Science and Engineering
2016/2017

Bachelor Thesis

Implementation of a scalable communication mechanism for MPI

Andrés Manglano Cañizares

Tutor
David Expósito Singh

Contents

List of Figures

Acknowledgements

A huge thank you to my amazing tutor, David Expósito Singh, for offering me the opportunity of developing this project and help me whenever I needed to.

To my parents, Mª Carmen and José Andrés, for a lifetime of support and love.

To my university partners with whom I could have not arrive to where I am now, specially the ones who have helped me with the countless projects we had to develop along these years.

To my partner, Sandra, who taught me to walk when you can not run, to crawl when you can not walk, and to find someone to carry me when you can not even crawl.

To all my friends who were always there when I needed a beer at the end of the day when working on this project.

Thank you.

Abstract

This document shows how the development of a scalable communication mechanism for MPI has been carried.

This work presents a communication mechanism applied to the monitorization of parallel distributed applications which has served as an example of how these kind of communication mechanisms work.

In order to implement the proposed system, in which an application spans a variable number of processes, a network communication topology shaped as a binary tree has been created. This allows the communications to be carried out in a scalar way which avoids network bottlenecks. In addition to that, a filtering mechanism in the communication has been implemented, which discards some messages which are irrelevant in the monitorization of the application.

This project has been developed in C language, using the MPICH implementation of the MPI standard for using parallel applications and the EVpath library for the communication mechanism.

Chapter 1

Introduction

Parallel distributed computing has always presented challenges, both to designers, developers and maintainers. Problems like network scalability, memory management or performance has provoked more than one headache to all the people involved in these kind of systems. In order to address these, mechanisms had and have to be researched and developed in order to increase the performance of these systems without consuming most of the resources available.

1.1 Motivation

Network communications and computing power of processors have evolved quite fast during the last years, making the usage of parallel distributed architectures very efficient as they can solve high computational problems in efficient ways.

Nevertheless, despite of these advances which for sure ease the work, these kind of problems grow more challenging with time. Problems like the aggregation of messages in a distributed application require scalable techniques in order to take advantage of all these mentioned advances. That is, with a great hardware infrastructure, a great software to use it must also come.

In this particular scenario, the problem comes in a High Performance Computing Cluster with many nodes running a software developed using the Message Passing Interface. This environment will be explained in the following section.

1.2 Investigation group context

The Computer Architecture and Technology Area in the Computer Science and Engineering Department, ARCOS, has a High Performance Computing Cluster which allows them to develop parallel distributed applications for investigation and research purposes. As this project will run inside this cluster, taking advantage of the Message Passing Interface, these components will be presented next.

1.2.1 Software environment: MPI

MPI, the Message Passing Interface, is a standard specification for implementing a message passing protocol to communicate processes which is widely used to communicate cluster nodes. It allows to design applications which span processes in different nodes of a cluster which are able to communicate among them as if they were in the same computer.

1.2.2 Hardware environment: Tucán

Tucán is the High Performance Computing Cluster of the ARCOS investigation group. It has 32 nodes distributed in three racks. It uses switches for either external and internal communications. The one connected to the exterior works at 100 Mbps, the ones which communicate nodes in a rack work at 10 Gbps and the one which communicate towers work at 1 Gbps.

Some nodes have GPUs incorporated for the use of parallel GPU computing using CUDA.

The different nodes are interconnected using the SSH protocol. The name of the nodes follow the pattern 'compute-X-X', so, to access one node, the following command must be executed in the terminal:

```
$ ssh compute-X-X
```

The file system installed in which the '/home' folder of the users is mounted in the cluster is NFS (Network File System). In addition to this, some nodes count with solid state storage which can be accessed in the mount point '/ssd'. Also, all the nodes count with a GLUSTERFS file system which can be accessed through the mount point '/mnt/gluster/hpc'.

1.3 Project Objectives

The project main objective resolves around the following problem in parallel distributed computing:

An application runs N processes in different nodes of a cluster. Each process performs some computation and it is required to monitor their performance. This performance is measured by the time a process spends in running an iteration of one internal loop of the application. These measurements have to be aggregated by the main process of the application. This is achieved by every process sending the measured time to the main process whenever it is measured.

The problem resides in the lack of scalability at the aggregation of the data by the main process. This schema creates a network bottleneck as the main process would have to deal with a request from each process each time it arrives.

To illustrate the problem, the following figure is provided in which an example of this kind of schema of execution with different processes running in different nodes. In this we can clearly see the scalability problem of aggregating the data directly in one node which receives data from all the nodes in the schema. Even we can appreciate that the problem is even worse when receiving data from different nodes, as the speed of transmission among nodes is different than the speed inside the nodes.

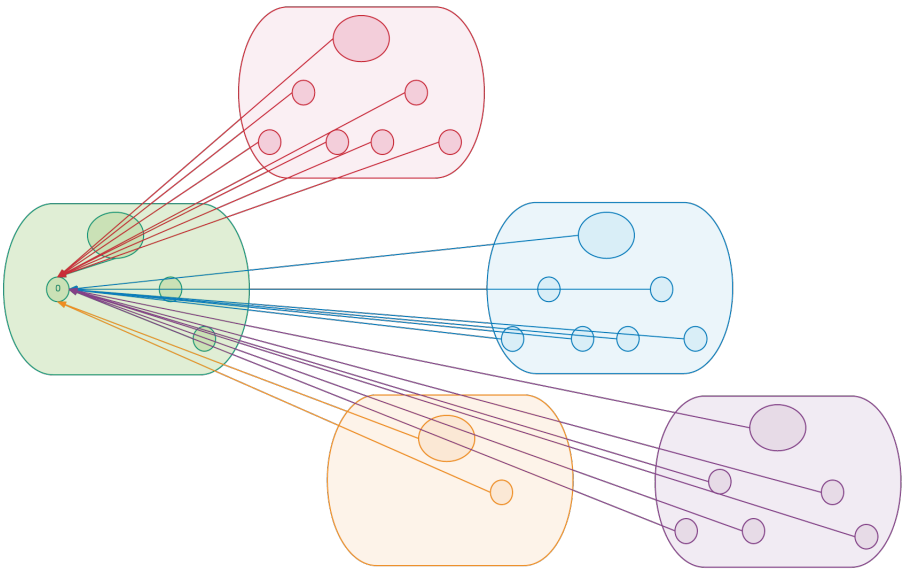

Figure 1.1: Problem description figure

In order to overcome this problem, the following objectives are stated:

- **Scalable data communication with low overhead**: A mechanism in order to communicate the processes distributed in nodes in a scalable way that avoids the network bottleneck explained above without adding much overhead, is part of the main objective of the project.
- **Conscious of the topology of the machine**: In order to accomplish the objective above, the developed project must be aware of the hardware configuration it would run on, in order to establish a topology mechanism which would let the communication mechanism to send the monitoring data in an efficient way.

In order to evaluate the performance of the developed project, two kind of tests are performed. One in which the throughput of the system is tested for a different number of nodes and a different period in the sending of messages, and another one in which the filtering mechanism is compared in the tree-based network architecture developed against a centralized architecture like the one explained in the figure 1.1.

1.3.1 Structure of the document

The structure of the document consist of six chapters and two appendices whose purpose will be explained now:

1. Introduction: In this chapter the description of the problem, the motivations and the objectives are presented.
2. State of the art: A study of the theoretical work around this topic is described in this chapter.
3. Development environment: This chapter presents the software considered and used in the development of this project is presented.
4. Proposal: In this chapter the objectives, requirements, design, planification, methodology and socio-economic context around the project are described.
5. Evaluation: A description on the hardware platform, the traceability matrix, the case studies proposed and the tests executed to measure the performance is stated in this chapter.
6. Conclusions and future work: The conclusions drawn after the development of the project are presented in this chapter alongside a description of how the work beyond this project could go in the future.

- User manual: In this appendix a guide for users to execute the project is stated.
- Developer manual: In this appendix the installation details for using the proposed project are explained.

Chapter 2

State of the art

This chapter provides an overview of the theoretical background related to the work performed in this project and around it. First, the currents challenges in supercomputation are presented, focusing on the Exascale computing. Second, the recent work in application monitoring is described. Third, a description of different communication tools in distributed systems is listed. Finally, the common problems with monitoring tools are described.

2.1 Current challenges in supercomputation: Exascale

In supercomputing, the major milestones are the building of systems that execute a number of floating operations per second (flops) above 10^{3k}, being "k" any number. The Gigascale (10^9), Terascale (10^12) and Petascale (10^15) thresholds were overcome in the past. [1]

The Exascale is a kind of computation system that overcomes the previous thresholds and also the new one of 10^18 floating points operations per second. This threshold is estimated to be surpassed by 2018.

In order to achieve that, a great amount of kinds of parallelism will have to be used. For that, the node architectures envisioned before will have to change significantly, as power and cooling constraints limit the microprocessor clock speeds. As a consequence, computer companies are increasing on-chip parallelism, such as, instead of increasing the clock speed, which as of now is constrained, they are increasing the number of cores the processors have to increase the number of threads they can execute.

Another kind of challenge to overcome will be the integration between the hardware and the software designed for it, as new algorithms will have to be developed in order to adapt to these new node architectures and take advantage of them.

Finally, the last, but not the least challenge to solve for these kind of systems are memory accessibility and speed, as for example, these kind of systems' hardware will have to deal good with locality and resilience. [2]

2.2 Recent work in application monitoring

In this section, some popular tools used in application monitoring are presented.

2.2.1 Periscope

Periscope is a performance analysis environment designed to analyze the performance of full size application runs. It allows to monitor the performance of grand challenge applications which run under supercomputers with up to hundreds of teraflops in an scalable way by performing an automatic distributed analysis over the network of the system. This help developers to monitor the performance of their applications which allow them to be tuned for better performance. [3]

2.2.2 Intel VTune Amplifier XE

Intel VTune Amplifier XE is a performance measurement tool developed by Intel which supports applications written in C, C++, C#, Java, Fortran and assembly. [4] It helps measuring the performance of single nodes in parallel applications based in different solutions like Pthreads, MPI or OpenMP, offering the programmers information about which parts of the code create bottlenecks and which ones consum more CPU power. [5]

2.2.3 Scalasca

Scalasca is a performance analysis set of tools designed for parallel applications which execute on large-scale systems with the order or thousands of processors. It provides two different performance analysis procedures, one which analyzes the runtime of the application and creates a summary of it and another which studies in depth the concurrent behavior of the application. It supports the analysis of MPI,

OpenMP and other kind of applications used in High-Performance Computation applications written in C, C++ or Fortran. [6]

2.2.4 Vampir

Vampir is a toolset designed to analyze the performance of parallel applications based on MPI. [7] It consists of two main components: the VampirTrace measurement system, which is the one in charge of the run-time monitorization, and the VampirServer, which is a X Window Server based visualization environment which allows the programmer to see where the monitored application spends more time running using its powerful zooming tool which allows the identification of problems at any level of detail. [8]

2.2.5 TAU

TAU (Tuning and Analysis Utilities) is a parallel performance system toolset and framework which lets programmers to measure, analyze and visualize performance information of large-scale parallel computer systems and applications.

It works following an architecture organized in three layers: instrumentation, measurement and analysis. The first allows the user to place performance instrumentation functions inside their C, C++, Fortran, Java or Python application which follow the measurement API. The latter layer, the analysis one, incorporates a visualization tool called ParaProf which is in charge of showing the monitorization results on the screen in a 3D environment. [9]

2.2.6 Paraver

Paraver (PARAlell Visualization and Events Representation) [10] is a performance tool for parallel applications running MPI or PVM architectures which helps to analyze and visualize parallel events traces. It provides a graphical visualization which works on the X Window system and allows the programmer to analyze the application tracing data obtained with Paraver.

This procedure consists of the following three steps:

- Data collection: Paraver is used to collect information about the execution of the application.
- Data analysis: The data collected is ordered and filtered and some statistics are calculated

- Display: The result provided in the analysis step is shown.

2.2.7 PAPI

PAPI is a specification of cross-platform interface to access to hardware performance counters. These hardware performance counters are implemented in modern microprocessors and are a small set of registers which count hardware events of the processor. The PAPI specification has a standard set of events relevant for application performance measurement and tuning, and also a set of both high-level and low-level routines for accessing these performance counters. [11]

This specification supports a great number of platforms, including Linux for 32 and 64 bits.

2.3 Communication tools in distributed systems

In this section, an overview on various different tools and technologies used for communication in distributed system is provided.

2.3.1 RPC

"Remote Procedure Call, is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel." [12]. This means, that a Remote Procedure Call is when a program executes a function which is in a different machine across the network. This call is transparent to the caller and it is executed as if it were a local call.

The fundamental idea behind the RPC model is that the server process exports an interface of procedures or functions that can be called by client programs. The clients make local procedure calls as if they were directly linked with the server, while under the covers, the procedure call is converted into a message exchange with the server process.

2.3.2 RMI

RMI (Remote Method Invocation) is a Java framework that implements an RPC mechanism [13] for the Java programming language which allows different application components to communicate via remote object invocations. More in detail, a client which runs in one node can access methods of a remote service by invoking a method of the object implemented by the service. That is, instead of exploiting the low level message passing technology used in RPC systems for their communication needs, RMI exploits the distributed object technology. [14]

2.3.3 CORBA

The Common Object Request Broker Architecture (CORBA) is a specification detailed by the Object Management Group (OMG) to implement a RPC mechanism. As RMI, CORBA is also object based and the calls are performed to objects in remote machines. While in RMI all the components in the system had to be Java based, in CORBA the different parts of the system that use the CORBA model do not have to be object oriented. [15]

2.3.4 WebServices

"A web service is a software system designed to support interoperable machine-to-machine interaction over a network." [16] It needs to be compliant with the Internet standards as it must offer a published interface that can be invoked across the Internet.

Web services are identified by an URI and their interfaces and bindings are able to be defined, described and discovered as XML artifacts, as they support interactions with other software agents by the usage of XML-based messages exchanged via Internet-based protocols. [17]

Chapter 3

Development environment

This chapter provides an overview of the different software libraries and environments used in order to develop the presented proposal. First the software alternatives studied to develop parallel applications are presented. Then the chosen library for establishing the communication mechanism, EVpath, is introduced. Then the considered benchmarks are described and finally, the different third party software employed in the development of the system is described.

3.1 Software alternatives to develop parallel applications

In the design of parallel applications, three main paradigms exist: shared memory architectures, distributed memory architectures and hybrid architectures.

The shared memory architectures are the ones in which different processes running in the same or different processors can access simultaneously to the same physical memory. This is typically used for communication among processes and/or threads of a single or different processes.

The distributed memory architectures are the ones in which the different processors has its own private memory and if one process requires data stored in a remote node of the architecture, network communication between the nodes is required, thus, a network overlay is implemented.

Hybrid architectures bring together characteristics of both shared and distributed memory worlds. In this architecture, each node has its own private memory but are also interconnected through the network.

In order to implement the proposed system as a parallel application, some alternatives were taken into account which will be explained:

3.1.1 MPI

MPI was originally designed for distributed memory architectures but as times changed, distributed memory architectures were combined with the shared memory ones, thus creating the hybrid architecture. This made MPI developers adapt their implementations to handle both memory models so that today MPI runs on any hardware configuration.

MPI is a standard specification for message passing libraries written in C and Fortran. Its name is an acronym for Message Passing Interface, that is, it is not a library by itself, so that allows different implementations to exist.

MPI Implementations

As stated before MPI is just an standard interface specification which has been implemented by different groups of developers. All library implementations differ in some bits like the version and/or features of the standard supported. The most relevant implementations will be presented in the following lines:

- MPICH is an open source, high performance and portable implementation of the Message Passing Interface standard developed by the US-government Argonne National Laboratory and the Michigan State University in collaboration with some other partners, such as IBM, the Ohio State University or Intel. It is licensed under a BSD-like license and implements almost every feature of the MPI standard in its 3.0 version. It is available for a wide variety Operating Systems, such as Unix, Unix-like and Windows systems. It's objectives are to run efficiently in cluster based systems with high speed networks and to provide a modular environment for their partners to collaborate with modules.
- OpenMPI is another open source implementation of the Message Passing Interface licensed under the BSD license. Their main goals are the creation of a high performance, free, open source, peer-reviewed MPI implementation which encourages input from the High Performance Computation community in order to avoid forks of the code by third party developers. As MPICH, it also implements all the MPI standard in its 3.0 version.
- Many other implementations exist including commercial ones by HP, Microsoft or Intel.

For this project, only the free and open source implementations have considered to reduce costs and legal issues. Between the two alternatives studied (MPICH and OpenMPI) MPICH has been chosen, as it is one most suitable to implement a parallel system in a Cluster-based architecture as the one we will be using.

3.1.2 OpenCL

OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of heterogeneous systems, such as personal computers, mobile devices or servers developed by the Khronos Group. It allows programs to take advantage of GPUs computing power. It is a competitor for Nvidia's own solution, CUDA, but with the advantage of being an open standard.

It provides two programming languages which are based on a subset of the C99 and C++14 standards respectively. These programming languages are adapted to fit the memory model of OpenCL and have lots of extensions whose purpose is to help with the use of parallelism.

3.1.3 MapReduce

MapReduce, as MPI, is a standard specification for parallel processing of large amounts of data in distributed memory environments, such as a cluster.

It follows a procedure composed of two parts:

- A Map function which receives a pair of key-value data on each iteration and returns a list of these key-value pairs, filtered and sorted by a given criteria.
- A Reduce function which is executed in parallel for each element of the key-value lists, which produces a smaller set of data to process.

The aim of map reduce is to provide this standard specification for parallel processing of data providing redundancy and fault tolerance.

The most relevant implementation of this specification is Apache Hadoop, which is an open source implementation developed by the Apache Foundation under the Apache License and which is written in Java language.

3.2 EVpath

EVpath is a an open source event transport middleware layer library released under the BSD License. It is developed by the Korvo Group of the public research university of Georgia Institute of Technology in Atlanta, Georgia.

It is a constructor library for building different types of messaging systems. It can be used to create publisher/subscriber systems, but it is not constrained to that, as it is designed to implement easily overlay networks with data processing, routing and management in all of its points.

EVpath is built around "stones". This stones are a concept that represent the starting and end point of a path over an overlay network. They can be compared to end points in data flow diagrams. This means, programmatically, that an EVpath stone can be understood as a process, or a thread, which acts as a transmission point in a communication system that can also be used to filter, transform, mux or demux data. The path is not a construct in EVpath but the concept rather exists as two or more connected stones in the overlay which establish a flow of data.

EVpath counts with the following seven types of stones:

- **Terminal stones**: These are the stones related with the receiving side of a communication in EVpath. They represent passive points of a communication, and are associated with an action represented by its stone ID and a contact list, which is a unique string that represents the information about how to establish a communication with the stone. Whenever they are contacted, they fire an action specified as a function which acts as an event handler. The actions associated with these stones are stated with the function *EVassoc_terminal_action*.
- **Bridge stones**: These are the ones related with the sending side of a communication in EVpath. Bridge stones are the ones which act as active points of a communication. They need the ID of a terminal stone and its contact list in order to establish an action that acts as a bridge of communication over the network. The actions associated with these stones are stated with the function *EVassoc_bridge_action*.
- **Split stones**: These are the ones related to actions which replicate the data stream that arrives to a stone in order to send it to a different kind of stone, which would act as a target. The actions associated with these stones are stated with the function *EVaction_add_split_target*.
- **Filter stone**: These stones are associated with actions that filter the data to be sent through a bridge stone making possible to dicard some data which may not be relevant for our overlay. The actions associated with these stones are stated with the function *create_filter_action_spec*.

- **Transform stones**: These stones are similar to the previously explained split stones, but slightly more complex, as rather than filter the data, they can also transform it, even modify its data type. The actions associated with these stones are stated with the function *create_transform_action_spec*.
- **Router stones**: These are the ones which, similarly to a split stone, are associated with an action that replicates the data stream, but in addition to that, this kind of stone can also output the stream of data to different and specific stones, which act as different targets. The actions associated with these stones are stated with the function *EVaction_set_output*.
- **Multi stones**: This are the most complex stone types in EVpath as they allow the creation of stones associated with more than one action. They also allow for the processing of different kinds of data types specifying preconditions which have to be met in order for the data to be processed. The actions associated with these stones are stated with the function *create_multityped_action_spec*.

EVpath makes use of functions and declarations from some other libraries which are its dependencies. For describing the kind of data structures which are to be transmitted over the overlay, and to describe the data types an event handler is to deal with, the library FFS is used. This library is designed to marshall and unmarshall data for high performance systems. The snippet 3.1 shows an example of these data types.

```
1  typedef struct perfInfo {
2      double time;
3  } PerfInfo, *PerfInfo_ptr;
4
5  static FMField PerfInfo_list[] = {
6      {"time", "integer", sizeof(int), FMOffset(PerfInfo_ptr,
           time)},
7      {NULL, NULL, 0, 0}
8  };
9
10 static FMStructDescRec PerfInfo_format_list[] = {
11     {"performance", PerfInfo_list, sizeof(PerfInfo), NULL},
12     {NULL, NULL}
13 };
14 %\label{code:Code1}
```

Listing 3.1: Example of FFS data types used in EVpath

For dealing with encoding and decoding of data strings, it uses functions such as *atl_base64_encode* and *atl_base64_decode* from the atl library.

EVpath also counts with a subset of dynamically-generated C language which is used in the filter and transformation stones called CoD (C-on-Demand) which is also part of the FFS library. CoD code is a text string which represents a function

whose input and output parameters are specified by the data types associated with the stone in which the filtering/transformation is to be performed. These strings are codified in base 64 using the atl functions stated above in the receiving parts of the communication, and sent through the network alongside the ID of the stone and the contact list associated to it, to the sending part of the communication, which is the one in charge of decoding the CoD function and to execute it to apply the filtering or transformation of the data. The snippet 3.2 shows an example of CoD function.

```
char *filter_func = "\
{\
    static double avg_dt = 0.0;\
    double a = avg_dt - input.time;\
    if(a < 0) {\
      a = a * -1;\
    }\
    if(a / avg_dt > 0.05) {\
      avg_dt = 0.8 * avg_dt + 0.2 * input.time;\
      return avg_dt;\
    }\
}";
```

Listing 3.2: Example of CoD filtering function

3.3 Description of considered benchmarks

In this section, the benchmark considered for the realization of the performance tests is introduced.

3.3.1 Jacobi

The Jacobi method is an iterative algebraic algorithm [18] which performs sequential computation by solving a system of linear equations. As it is, it provides a way to measure the time it takes for a system to compute the solution of the algorithm, as the execution time is proportional to the square of the size of the matrix.

This method is really useful in order to perform benchmarks in distributed and parallel applications, as with the division of the problem in different processes, the execution time decreases proportionally with the number of processes.

For our purpose, we employ a modified version of the Jacobi benchmark which makes use of MPI for performing the benchmark in a distributed environment. It

adds communication among processes intensive operations and also, I/O intensive operations.

These operations can be adjusted by setting their complexity as program arguments. An example of execution of the benchmark would be:

```
$ mpiexec -np 8 -f rankfile ./jacobi 500 1000 0.00001 1 1 1
```

Being the program arguments the following:

- **-np**: This is an mpiexec flag to set the number of processes which would run the program.
- **8**: The number of process.
- **-f**: Another mpiexec flag to specify a rankfile.
- **rankfile**: The path of the rankfile to configure mpiexec. This is a plain text file which sets the number of processes to be run by each node of a cluster.
- **./jacobi**: The path of the binary of the Jacobi benchmark.
- **500**: The matrix size, which, as said before, adds complexity to the code in a factor of approximately the square of its value.
- **1000**: This is the number of iterations to be executed by the main loop of the program.
- **0.00001**: This is the convergence criteria, which state how the numbers in the matrix are generated. If this criteria fails, and the algorithm does not converge, it would end once it reaches the number of iterations stated in the previous argument, which is the worst case scenario for this.
- **1**: The CPU complexity. It establishes how many times per iteration, a kernel computation is run. If set to 0, no CPU computation is executed.
- **1**: This is the communication complexity, which establishes how many times per iteration a communication operation is run. If set to 0, no communication is executed. This communication operation is a *MPV_Allgatherv* which make all the processes to send data among them to be gathered by every one of them.
- **1**: The I/O complexity, which establishes how many times per iteration an I/O operation is run. If set to 0, no I/O computation is executed. The I/O operation executed is a write to a file

The modification made to the method in order to perform communication operations conflicts with the idea of the original Jacobi benchmark in distributed applications, as executing the function *MPV_Allgatherv* on every iteration make all the processes to perform communications among all of them. This creates an execution delay proportional to the number of processes stated as execution argument, as with a higher number of processes more communications are needed to gather the data from all of them, hence a higher execution time is taken by Jacobi to finish.

3.4 Development environment

In this section the different Operating Systems, libraries, dependencies, and other tools used in the development of this project will be detailed.

3.4.1 Operating systems

KDE Neon

For the development of the code of the project KDE Neon was the chosen one. This OS is a GNU/Linux distribution based in the latest LTS (Long-Term Support) version of Ubuntu (16.04 as this is written) developed by the KDE team. This team is the one behind the Plasma desktop environment and KDE applications, so this distribution is always updated with the latest software from the KDE team and also with the stable base of an Ubuntu LTS release. The decision to use this operating system was made based on the fact that this is the primary operating system in my PC, so it is the one I am used to as all my projects, the personal ones and the ones for University, are developed in this operating system. Also it seemed suitable for the development of the project, as this operating system is based in the one running in the Tucán cluster.

Ubuntu Server 16.04.2 LTS

This is the operating system used by the nodes which comprise the Tucán cluster. This is a Debian based GNU/Linux distribution developed by Canonical on its server version. As the election of the OS of the cluster is not a task for the developer of this project, but rather to the system administrator there was no alternative for this software.

3.4.2 External Libraries

Next, the libraries used by the project are presented.

EVpath

EVpath is a library aimed to act as an event transport middleware layer. In this project, it was used to deploy an overlay network among the different processes, which make them able to establish communications among the different processes and to process the data exchanged in these communications.

MPI

MPI (Message Passing Interface) is, as stated before, a library used to develop parallel computing applications. In this project it has been used in order to create the different processes which make up a node in the overlay. For this project, the MPICH implementation has been used in its latest stable release, the version 3.2.

MPI was chosen among the different alternatives presented before, as it is the one which meets the requirements better. That is, MPICH provides a way to develop a parallel application written in C language, which would run in a cluster running on top of a Linux distribution. A framework to use GPU capabilities for parallelism is not needed, so OpenCL does not seem like the optimal option. Neither a framework to manage big amounts of data is needed, so MapReduce could be discarded too.

3.4.3 Other software

Kate

Kate (KDE Advanced Text Editor) is the default text editor for the Plasma Desktop. This is a rich feature text editor, which was used to write the code of the project, due to this. Among its useful features, some of them stand out, like code completion, dynamic word wrap or tabulator and matching brackets highlighting.

Konsole

Konsole is the default terminal emulator for the Plasma Desktop. It is a featured rich console which offers some functionality as tabbed terminals or customizable key bindings.

Vim

Vim is a popular Unix text editor which is used from a command line. It has been used in order to modify configuration files and source code files of the project which reside in the Tucán cluster as it does not provide with a user interface.

Valgrind

Valgrind is an open source set of tools which help to debug applications for memory usage and performance. The most popular tool, which was used in the development of the project, is Memcheck, which helps detecting memory errors, like leaks or usage of non initialized memory. This tool was chosen, as it is one of the best tools for debugging C code.

SSH

SSH or Secure SHell is a protocol used over the network in order to access remotely other computer systems through a secure connection. It provides total access to the host computer with transmission of files included. It has been used to access the Tucán cluster, as it provides a SSH server.

GCC

This is the standard compiler system developed by the GNU project. It is the compiler used for projects written in C in most standard Unix/Linux implementations and also the one used as a backend for the MPI compiler MPICC.

Chapter 4

Proposal

This chapter has the purpose of describing the project developed as a thesis.

4.1 Objectives, work methodology and development phases

In this section, the objectives to achieve in the project, the work methodology followed and the development phases through which it passed are presented.

The objectives of the project where specified in a meeting at the beginning of it. The main objective was fixed: the solution to the problem of scalability in a monitoring system in a distributed environment which was presented in the introduction of this document. 1

In order to carry with the development process of the project, it will follow a Waterfall Model. This was created by Winston W. Royce in 1970 [19]. This is an sequential model where a phase is not started if the previous one is not finished. This model was chosen over others like the V-Model or evolutionary development as this project has stable requirements and the fact that one phase has to be completed before the next one is started suites the development of this project well.

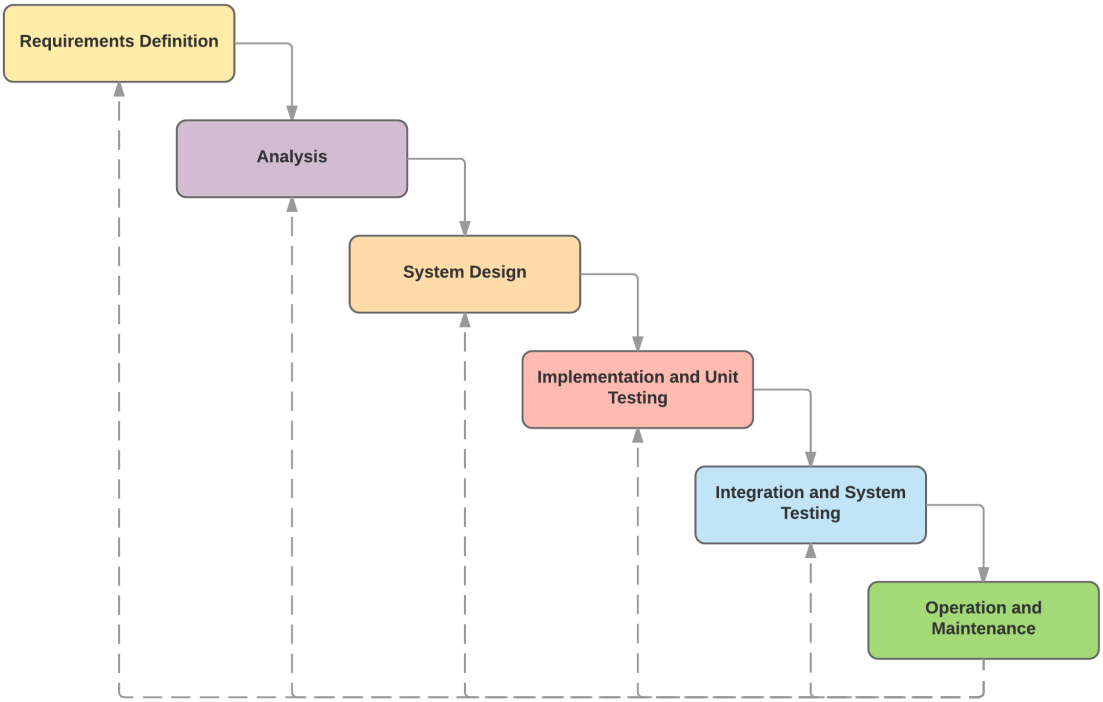

Figure 4.1: Waterfall development model

The different phases are described:

- Requirements Definition: This phase consists of the analysis of the problem to be solved in order to determine the characteristics the system is supposed to have. For this, use cases are determined in order to obtain the different requirements, both user, functional and non-functional.
- Analysis: In this stage the software to be used is analyzed in order to obtain a good knowledge of their functionality to be able to take advantage of them.
- System and Software Design: In this phase, the previously stated requirements are used in order to create a design of the components of the system.
- Implementation and Unit Testing: In this stage the code is developed.
- Integration and System Testing: In this phase the components of the system are put together. Then it is tested in order to know if its behavior is correct.
- Operation and Maintenance: In this phase the system in installed and deployed in the production system and supported. In this case, the documentation has been finished.

4.2 Use cases analysis

In this section, the use cases covered by the project are stated.

The following figures show a representation of the different use cases.

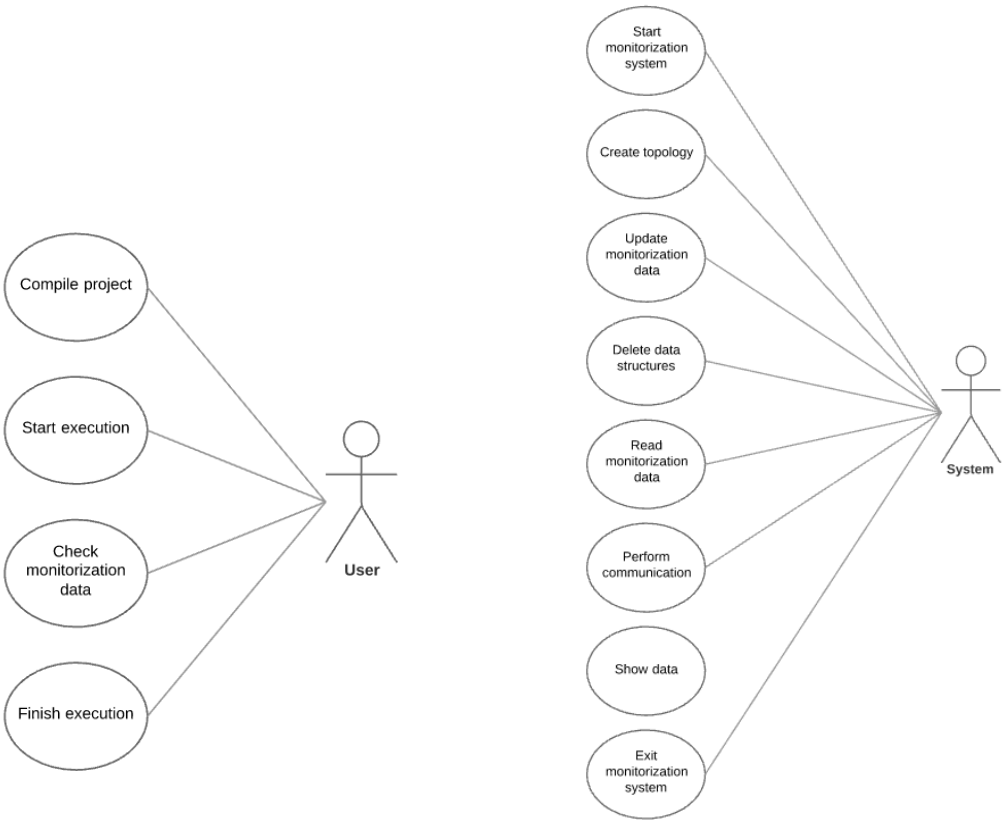

Figure 4.2: Use cases diagrams

The attributes of each use case will be specified using a table like the following.

Use case	
ID	UC-XX
Name	Name of the use case
Actors	Actor involved in the use case
Description	Description of the use case
Preconditions	Preconditions of the use case
Effects	Effects of the use case
Scenario	Scenario in which the use case takes place

Table 4.1: Use case UC-XX

The meaning of each attribute is described below:

- ID: Unique identifier of the use case. It is formed by the use case's code UC followed by a "-" and two digits.
- Name: Identification name of the use case.
- Actors: Identifies the entities involved in the use case.
- Description: Basic description of the use case.
- Preconditions: Conditions that need to be true before the execution of the use case.
- Effects: Functionality executed by the system when the use case is performed.
- Scenario: Description of the actions executed in the use case.

The following tables show the description of each use case.

Use case	
ID	UC-01
Name	Compile project
Actors	User
Description	The user compiles the project
Preconditions	None
Effects	The system is compiled, the binary executable is generated
Scenario	The user compiles the project using the Makefile. An execution binary is generated by the compiler

Table 4.2: Use case UC-01

Use case	
ID	UC-02
Name	Start execution
Actors	User
Description	The user starts the execution of the system
Preconditions	The system must be previously compiled
Effects	The execution of the system starts
Scenario	The user executes the executable binary file generated and the execution of the system starts

Table 4.3: Use case UC-02

Use case	
ID	UC-03
Name	Check monitorization data
Actors	User
Description	The user checks the monitorization data shown in the screen
Preconditions	The system must be running
Scenario	The system is executing and the user can see the data shown by the monitorization system

Table 4.4: Use case UC-03

Use case	
ID	UC-04
Name	Finish execution
Actors	User
Description	The system finishes execution and exits
Preconditions	The system must be running
Effects	The system stops
Scenario	The system has finished all of its execution and exits

Table 4.5: Use case UC-04

Use case	
ID	UC-05
Name	Start monitorization system
Actors	System
Description	The monitorization mechanism is executed by the system
Preconditions	The system must be started
Effects	The monitorization mechanism is launched in all the nodes of the system
Scenario	The system executes the monitorization mechanism by invoking its functionality in every node of the system

Table 4.6: Use case UC-05

Use case	
ID	UC-06
Name	Create topology
Actors	System
Description	The tree-shaped network topology is generated by all nodes of the system in order to perform the communication protocol
Preconditions	The monitorization mechanism must be started
Effects	The network topology is created in all the nodes of the system as two balanced binary tree data structures assigning memory to them
Scenario	The monitorization system is executing and in order to establish the communications, a model of the network is created in all the nodes.

Table 4.7: Use case UC-06

Use case	
ID	UC-07
Name	Update monitorization data
Actors	System
Description	The data to be monitored is updated by the system
Preconditions	The system must be running
Effects	The monitorization data is updated
Scenario	The system is running and the monitorization data is updated periodically

Table 4.8: Use case UC-07

Use case	
ID	UC-08
Name	Delete data structures
Actors	System
Description	The data structures in which the model of the network is stored are freed from memory
Preconditions	The system must be running, the monitorization mechanism too and the data structures which represent the network topology must be created created
Effects	Effects of the use case
Scenario	Scenario in which the use case takes place

Table 4.9: Use case UC-08

Use case	
ID	UC-09
Name	Read monitorization data
Actors	System
Description	The data to monitor is read in order to be sent
Preconditions	The data must be updated periodically
Effects	The data is read by the monitorization system
Scenario	The monitorization mechanism needs to send the monitorization data, so it reads the value first

Table 4.10: Use case UC-09

Use case	
ID	UC-10
Name	Perform communication
Actors	System
Description	The monitorization data is sent through the network to the corresponding nodes
Preconditions	The monitorization data must be read
Effects	The data is transmitted over the network
Scenario	The network topology has been established, the monitorization data has been read, and the sending nodes send it while the receiving ones wait for the data

Table 4.11: Use case UC-10

Use case	
ID	UC-11
Name	Show data
Actors	System
Description	The monitorization data is shown on the screen by the system
Preconditions	The monitorization mechanism must be running
Effects	The data is shown on the screen
Scenario	The monitorization mechanism is running and showing the relevant data on the screen

Table 4.12: Use case UC-11

Use case	
ID	UC-12
Name	Exit monitorization system
Actors	System
Description	The communication mechanism stops its execution and the monitorization mechanism exits
Preconditions	The monitorization mechanism must be running
Scenario	The system has finished its execution, so the monitorization mechanism has to exit too

Table 4.13: Use case UC-12

4.3 Requirements analysis

In this section, the requirements to be fulfilled by the system are described. First, a description of the information presented on each requirement is presented. Then the functional requirements are presented and finally the non-functional ones.

4.3.1 Requirements attributes

The attributes will be specified using a table like the following one.

User or System Requirement			
ID	UR-XX,FR-XX or NFR-XX	**Source**	Client or developer
Name	Requirement's name		
Description	Requirement's description		
Relation	Relation with other requirements		
Significance	Essential, desirable or optional	**Priority**	High, medium or low
Stability	Stable or unstable	**Verifiability**	High, medium or low

Table 4.14: Functional requirement XXX.

The meaning of each attribute is described below:

- ID: Unique identifier of the requirement. It is formed by the requirement's code UR (User requirement), FR (Functional Requirement) or NFR (Non-Functional Requirement) followed by a "-" and two digits.
- Name: Identification name of the requirement.
- Description: Basic description of the requirement.
- Source: Indicates from where the resource was identified.
- Significance: Determines in which degree the requirement must be implemented. The possible values can be:
  - Essential: The requirement must be implemented.
  - Desirable: It would be preferable to implement the requirement, but it is not mandatory.
  - Optional: The requirement can be implemented but it is not mandatory.

- Priority: It sets the importance of the requirement in order to be included in the development process. It can take the following values:
  - High: The requirement must be implemented in the initial stages of development.
  - Medium: The requirement must be implemented once the high priority requirements have been implemented.
  - Low: The requirement to implement will not impact in the correct operation of the system. It must be implemented once the medium priority requirements are.
- Stability: Defines if the requirement can be modified during the software life cycle or not. The possible values can be:
  - Stable: The requirement cannot be modified during the life cycle of the system.
  - Unstable: The requirement can be modified during the life cycle of the system.
- Verifiability: Defines in which degree is possible to check the requirement has been implemented in the system. It can take the following values:
  - High: It can be verified for the requirement to be implemented.
  - Medium: It can be verified for the requirement to be implemented but takes a complex verification.
  - Low: It is difficult or impossible to verify the implementation of the requirement.

4.3.2 User requirements

In this section the user requirements are listed. These requirements define the functionalities demanded by the client.

User Requirement			
ID	UR-01	**Source**	Client, UC-03 and UC-11
Name	The user must be able to see the data read by the monitorization system		
Description	The relevant execution time measures must be shown to the user		
Relation	FR-01		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	High

Table 4.15: User requirement UR-01

User Requirement			
ID	UR-02	**Source**	Client, UC-05 and UC-06
Name	The user must be able to execute the system in a cluster		
Description	The user must be able to execute the system on the Tucán cluster of the ARCOS research group		
Relation	NFR-02, NFR-03 and NFR-04		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.16: User requirement UR-02

User Requirement			
ID	UR-03	**Source**	Client
Name	The system must be centralized		
Description	The developed system must have a centralized data aggregator		
Relation	FR-05 and FR-08		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.17: User requirement UR-03

User Requirement			
ID	UR-04	**Source**	Client and UC-03
Name	The user should not see irrelevant monitoring data		
Description	The user should not be able to see on the screen data considered not relevant for the monitorization process		
Relation	FR-03		
Significance	Essential	**Priority**	Medium
Stability	Stable	**Verifiability**	High

Table 4.18: User requirement UR-04

User Requirement			
ID	UR-05	**Source**	Client, UC-01, UC-02, UC-03 and UC-04
Name	The user should not see any difference between executing the application with or without the monitorization system		
Description	The only difference the user should tell when executing the monitorization system is the monitoring data showing on the screen		
Relation	FR-02 and FR-04		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	High

Table 4.19: User requirement UR-05

4.3.3 Functional requirements

In this section the Functional requirements are presented. These are system requirements which define the system functionality.

System Requirement			
ID	FR-01	**Source**	Client and UC-11
Name	Monitoring data must be shown on the screen		
Description	The relevant execution time measures must be shown on the screen		
Relation	UR-01 and NFR-01		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	High

Table 4.20: Functional requirement FR-01

System Requirement			
ID	FR-02	**Source**	UC-02, UC-03 and UC-04
Name	Execution of the system must be transparent to the user		
Description	The steps to execute an application monitored by the system developed must be the same as executing one which is not monitored by the system		
Relation	UR-05		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	High

Table 4.21: Functional requirement FR-02

System Requirement			
ID	FR-03	**Source**	User
Name	Monitoring data must be filtered if not relevant		
Description	If the monitoring data read in a specific iteration does not variate at least a 5% from the previous data read, it should not be sent		
Relation	NFR-05 and UR-04		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	Medium

Table 4.22: Functional requirement FR-03

System Requirement			
ID	FR-04	**Source**	UC-04 and UC-12
Name	Finish of the monitoring system must be transparent to the user		
Description	The monitoring system must finish whenever the application monitored finishes its execution		
Relation	UR-05		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	High

Table 4.23: Functional requirement FR-04

System Requirement			
ID	FR-05	**Source**	UC-06
Name	Usage of a binary tree based topology		
Description	The communication topology must be arranged in a binary tree structure		
Relation	NFR-05 and NFR-06		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.24: Functional requirement FR-05

System Requirement			
ID	FR-06	**Source**	UC-07
Name	Data must be updated on every iteration		
Description	Execution time must be updated on every iteration of the main loop		
Relation	NFR-08, NFR-09 and NFR-10		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.25: Functional requirement FR-06

System Requirement			
ID	FR-07	**Source**	UC-09
Name	Data must be read and sent every second		
Description	Iteration execution time must be read and sent by each node of the overlay to its parent every second		
Relation	NFR-03, NFR-04 and NFR-05		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	Medium

Table 4.26: Functional requirement FR-07

System Requirement			
ID	FR-08	**Source**	UC-03
Name	Aggregation of data by the node whose ID is 0		
Description	The node whose ID given by MPI is 0 must be the one which aggregates the data transmitted through the network overlay		
Relation	UR-03		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	High

Table 4.27: Functional requirement FR-08

System Requirement			
ID	FR-09	**Source**	Client
Name	The data structures created must be destroyed upon communication		
Description	Before the data starts to be transmitted through the overlay, the data trees created to configure the communication must be taken out of the main memory		
Relation	NFR-07		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	Medium

Table 4.28: Functional requirement FR-09

4.3.4 Non-Functional requirements

In this section the non-functional requirements are listed. These are also system requirements which define characteristics for the system that are related to its operation.

System Requirement			
ID	NFR-01	**Source**	Client
Name	Use C language		
Description	The programming language used in order to implement the system is the C language		
Relation	FR-01 and UR-01		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.29: Non-Functional requirement NFR-01

System Requirement			
ID	NFR-02	**Source**	Client
Name	Use of EVpath library		
Description	The system must take advantage of the functionality given by the EVpath library in order to create a communication protocol		
Relation	FR-07		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.30: Non-Functional requirement NFR-02

System Requirement			
ID	NFR-03	**Source**	UC-10
Name	Use of MPICH library		
Description	The system must use the MPICH implementation of the MPI protocol in order to operate in a distributed environment		
Relation	UR-02 and FR-08		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Low

Table 4.31: Non-Functional requirement NFR-03

System Requirement			
ID	NFR-04	**Source**	UC-10
Name	Compatibility with distributed memory systems		
Description	The system must be functional in a distributed memory architecture		
Relation	UR-02 and NFR-05		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Medium

Table 4.32: Non-Functional requirement NFR-04

System Requirement			
ID	NFR-05	**Source**	UC-06
Name	Scalability		
Description	The communication protocol created for the system must be scalable in order to avoid network overhead		
Relation	FR-05 and FR-03		
Significance	Essential	**Priority**	High
Stability	Stable	**Verifiability**	Low

Table 4.33: Non-Functional requirement NFR-05

System Requirement			
ID	NFR-06	**Source**	UC-06
Name	The data complexity of the data structures used must be linear at worst		
Description	The data structures employed to generate the network topology must have a data complexity which grows linearly with the number of elements		
Relation	FR-05		
Significance	Desirable	**Priority**	High
Stability	Stable	**Verifiability**	Low

Table 4.34: Non-Functional requirement NFR-06

System Requirement			
ID	NFR-07	**Source**	UC-08
Name	Low memory consumption		
Description	The memory footprint of the system must be low		
Relation	FR-05 and FR-09		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	Medium

Table 4.35: Non-Functional requirement NFR-07

System Requirement			
ID	NFR-08	**Source**	UC-05
Name	The system must monitor parallel CPU intensive applications		
Description	The system must monitor distributed applications based on iterative benchmarks which implement CPU intensive computation		
Relation	FR-06		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	Medium

Table 4.36: Non-Functional requirement NFR-08

System Requirement			
ID	NFR-09	**Source**	UC-05
Name	The system must monitor parallel communication intensive applications		
Description	The system must monitor distributed applications based on iterative benchmarks which implement communication intensive computation		
Relation	FR-06		
Significance	Desirable	**Priority**	Medium
Stability	Stable	**Verifiability**	Medium

Table 4.37: Non-Functional requirement NFR-09

System Requirement			
ID	NFR-10	**Source**	UC-05
Name	The system must monitor parallel I/O intensive applications		
Description	The system must monitor distributed applications based on iterative benchmarks which implement I/O intensive computation		
Relation	FR-06		
Significance	Optional	**Priority**	Low
Stability	Stable	**Verifiability**	Medium

Table 4.38: Non-Functional requirement NFR-10

4.4 System design

In this section, the system design is explained in detail. First, the developed components are described in section 4.3.1. Second, the existing architecture of the processed to be monitored is analyzed. Third, the model developed to determine the aggregation and communication topology is explained in detail.

4.4.1 Developed components

For the realization of the project a main component have been developed, a monitoring thread, which is executed by each process in the topology of the application to monitor its execution.

Monitoring thread

The monitoring thread, as stated above, is the main component of the developed project and consists of the following modules:

- A mechanism to create a network topology with the different nodes in which the application will run.
- A communication mechanism which performs the data transmission among the nodes of the topology.

The type of applications for which our project would be monitoring the performance, are MPI ones which span one or more processes in different nodes of a cluster. Each process spanned, performs intensive computation inside a main loop. Due to time limitations, only one simple parameter is taken into account to monitor the performance of the applications: the time it takes to perform a loop iteration.

For the purpose stated above the following steps have to be performed in order to set up the monitorization of the application:

1. Include the "*evp.h*" header developed at the start of the code.
2. Start the monitorization module with a call to the function *monitor_init()* before the computation of the application starts. The *monitor_init()* execution flow is explained in figure 4.14.
3. Get the initial time at the start of the main loop with the function *iteration_init()*.The *iteration_init()* execution flow is explained in figure 4.17.

4. Get the finishing time of the iteration at the end of the main loop and calculate the time to compute the loop with the function *iteration_end()*. This function execution flow is explained in figure 4.15.

5. Finish the monitorization module with a call to the function *monitor_stop()*. This function execution flow is explained in figure 4.18.

In the code 4.1 an example of the previous configuration can be seen. The lines commented with a line starting with a number correspond to the elements listed above with the same number.

```
1
2  // 1. Include the "evp.h" header developed at the start of
       the code.
3  #include "evp.h"
4
5  int main (int argc, char *argv[]) {
6      int i = 0;
7      // Initialize MPI environment
8      int provided;
9      MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &
           provided);
10
11     // 2. Start the monitorization module
12     monitor_init();
13     // Main loop
14     for(i = 0; i < condition; i++) {
15         // 3. Get the initial time at the start of the main
               loop
16         iteration_init();
17
18         // Perform computation
19
20         // 4. Get the finishing time of the iteration and
               calculate the time to execute it at the end of
               the main loop
21         iteration_end();
22     }
23     // 5. Finish the monitorization module
24     monitor_stop();
25
26         // Finalize MPI environment
27         MPI_Finalize();
28 }
```

Listing 4.1: Example of project configuration

4.4.2 Algorithms: Analysis of existing architecture used by the processes

The architecture used by the processes is, as stated in the code example above and in the last chapter, an MPI application which executes in distributed way.

Every process launched by an MPI application have one id parameter called *rank* and and two environmental parameters called *processor_name* and *world_size*. The rank parameter is a unique id of the process running. It is an integer value which could be a number between 0 and world_size, being the latter the number of processes running in the MPI application. The processor_name parameter is just the name of the node the process is running in. This process layout is shown in figure 4.3

Figure 4.3: Parameters of an MPI process

If an MPI application is executed with 6 processes as program argument the topology of the processes would look like the figure 4.4

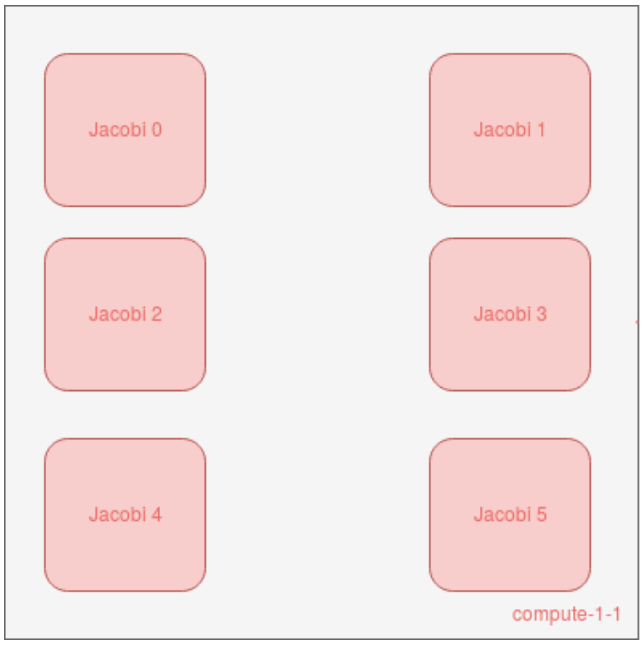

Figure 4.4: MPI application with 6 processes

For this project, the MPI application must be executed in different nodes of the Tucán cluster. In order to do this, a rankfile must be passed as an argument of the MPI application. This kind of execution would look like the figure 4.5

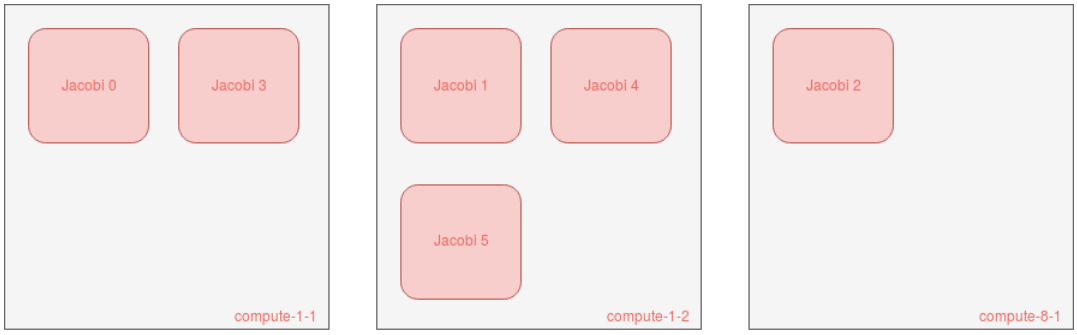

Figure 4.5: MPI application with 6 processes executing in different nodes

The rankfile would look like the example 4.2

```
1 node1:2
2 node2:3
3 node3:1
```

Listing 4.2: Rankfile configuration

4.4.3 Models: determination of aggregation/communication topology

The monitoring mechanism, as explained before, runs as a thread of execution associated to every process of the topology which is the one in charge of measuring the performance of that process and to communicate that measurements to other monitoring threads of other processes. This thread is launched whenever the function *monitor_init()* is called.

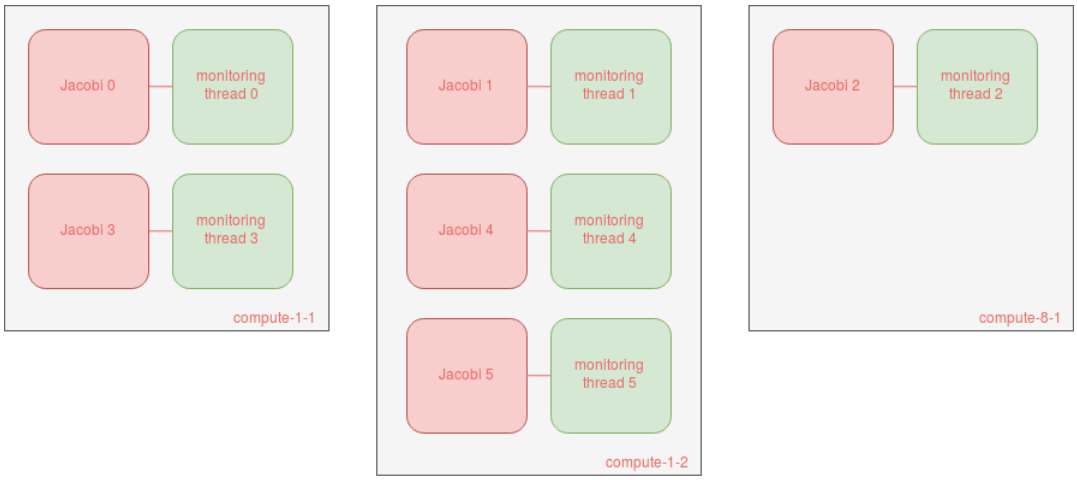

Figure 4.6: MPI application with monitoring threads running for each process

Then, the thread creates a network topology shaped as a binary tree, either for communicating the processes inside one node and also for communicating the processes from different nodes. An example of binary tree structure can be found in 4.7

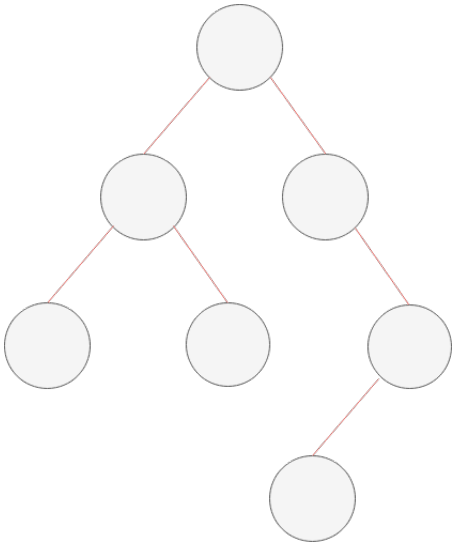

Figure 4.7: Structure of a binary tree

For creating this topology, two trees are created inside every process in the overlay. These trees are **Balanced Binary Search trees**. This means, that for every set of three nodes, the one with not the lowest neither the highest value is going to be the parent, while the one with lowest value is going to be left child and the one with the highest value is going to be the right child. An example of Balanced Binary Search Tree can be seen in figure 4.8

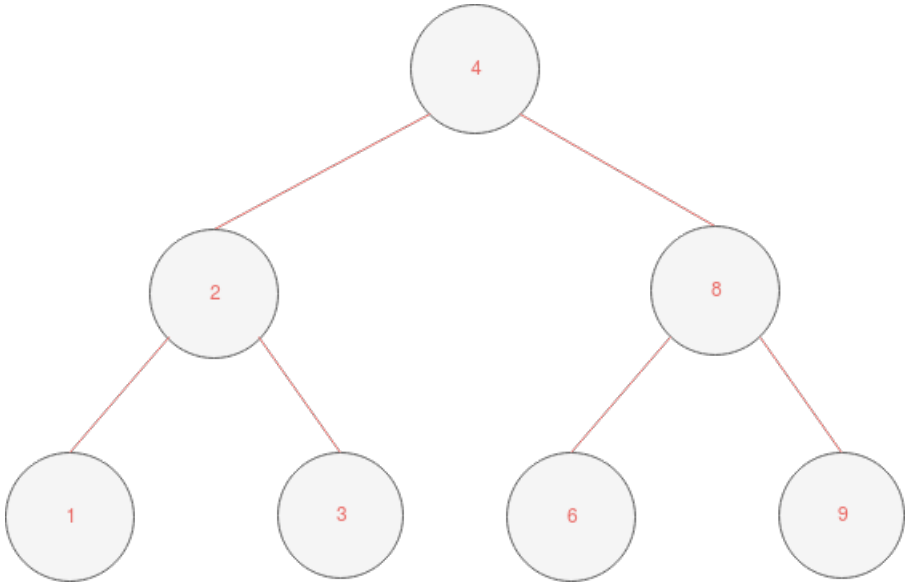

Figure 4.8: Example of a Balanced Binary Search Tree

The, previously mentioned, two Balanced Binary Search Trees which are to be created inside every process are the *Local Tree* and the *Global Tree*. The *Local Tree* is the one which states the communication of the processes inside the same node of the cluster, while the *Global Tree* is the one which states the communication between processes of different nodes of the cluster.

The *Global Tree* would be the same for all the processes, while the *Local Tree* would be the same for all the processes inside of a node of the cluster, while for the processes inside a different node, it would be different.

This means that the processes running inside a node of the cluster would form a *Local tree* of communication topology, ordered by the ranks of those processes, whose root node would aggregate the results of the monitorization of itself, and also the processes from its children nodes.

For the *Global Tree*, the roots of all the *Local trees* form another communication topology, again, ordered by the ranks of the processes. The root of this tree would receive the data from all its children and would send it to the node whose process rank is 0.

In order to create these two trees, first we obtain an array with the required parameters of each process in the application by using the MPI function *MPI_Allgather*.

The data to be gathered from each process is represented by a C struct named *Comm* with the following attributes shown in code 4.3

```
struct comm {
    int rank;
    char processor[2048];
    int recvInterLeft;
    int recvInterRight;
    int sendInter;
    int recvIntraLeft;
    int recvIntraRight;
    int sendIntra;
    int isRoot;
    int isGlobalRoot;
};
typedef struct comm Comm;
```

Listing 4.3: Information from each process of the topology

The attributes of each node are the following:

- **rank**: This is the ID of the process, as explained before.
- **processor**: This is the name of the node in which the process is running.
- **recvInterLeft**: This is the ID of the left child of the process in the *Global Tree*.
- **recvInterRight**: This is the ID of the right child of the process in the *Global Tree*.
- **sendInter**: This is the ID of the parent of the process in the *Global Tree*.
- **recvIntraLeft**: This is the ID of the left child of the process in the *Local Tree*.
- **recvIntraRight**: This is the ID of the right child of the process in the *Local Tree*.
- **sendIntra**: This is the ID of the parent of the process in the *Local Tree*.
- **isRoot**: This is a Boolean variable which states if the process is the root of a *Local Tree* or not.
- **isGlobalRoot**: This is a Boolean variable which states if the process is the root of a *Global Tree* or not.

At this point, all the attributes are set to -1 but the rank and the processor which are set following the MPI process attributes.

For explaining the functionality of the *MPI_Allgather* function, diagram 4.9 is presented.

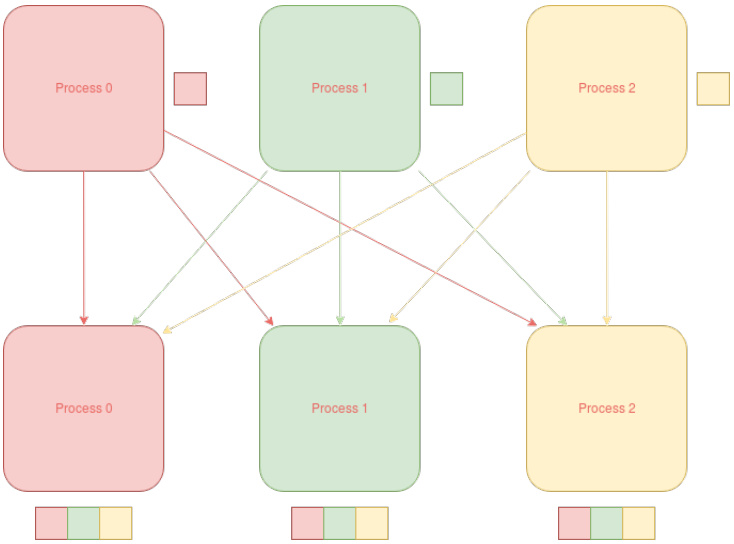

Figure 4.9: Example of MPI_Allgather execution

As we can see in the figure, with this function, all processes in the application share a portion of data, a Comm struct in this case, and gather all this portions from every process.

After having all this data gathered by every process, they immediately proceed to the creation of their corresponding *Local Tree*. In order to do this, they check all the data gathered, and create an array with the Comm structs of all the processes whose processor name is the same as theirs.

With this array, the function to generate the Balanced Binary Search Tree is called, thus obtaining the *Local Tree*. Next, the *walkIntra* function is called. This function would walk the whole tree, assigning the corresponding values to the parameters of each node of the tree.

Figure 4.10 explains how this works.

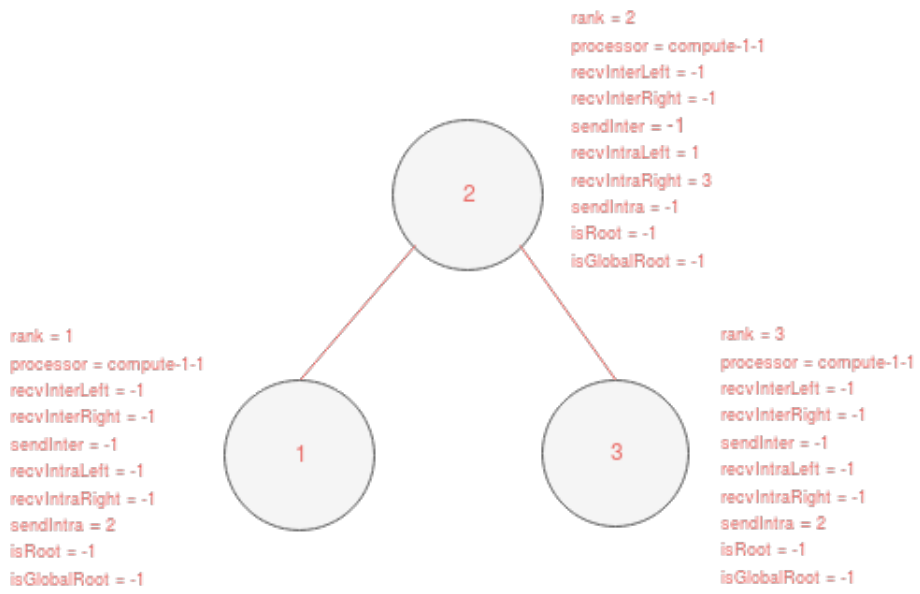

Figure 4.10: Example of walkIntra execution

Then, another *MPI_Allgather* execution is performed in order to update all the nodes in the topology with the information set by the execution of the *walkIntra* function. Doing this, every node has the information about from which nodes receive or to which nodes send everyone in the topology, thus being able to identify the root of every *Local Tree*.

With that information gathered, the root of every *Local Tree* of each node of the cluster form a *Global tree* of communication topology. The only exception for this is the case in which the process with the rank 0 is a root node of a *Local Tree*, in this case, that process would not be added to the *Global Tree*.

After all the *Local Trees* and the *Global tree* are formed, the root of the *Global Tree* is set to point to the process ranked 0, which is the one which should aggregate all the monitoring data.

When this process is finished, the network topology has been created, and every node knows it so they are prepared to become a specialized node in order to establish the communication. The configuration of the network topology execution flow is explained for the local trees in figure 4.27 and for the global one in figure 4.26.

Lets take the example of the problem stated in the introduction 1 with this mechanism applied, in which we can see how the ranked 0 node aggregates all the communication, is now shown for clarification.

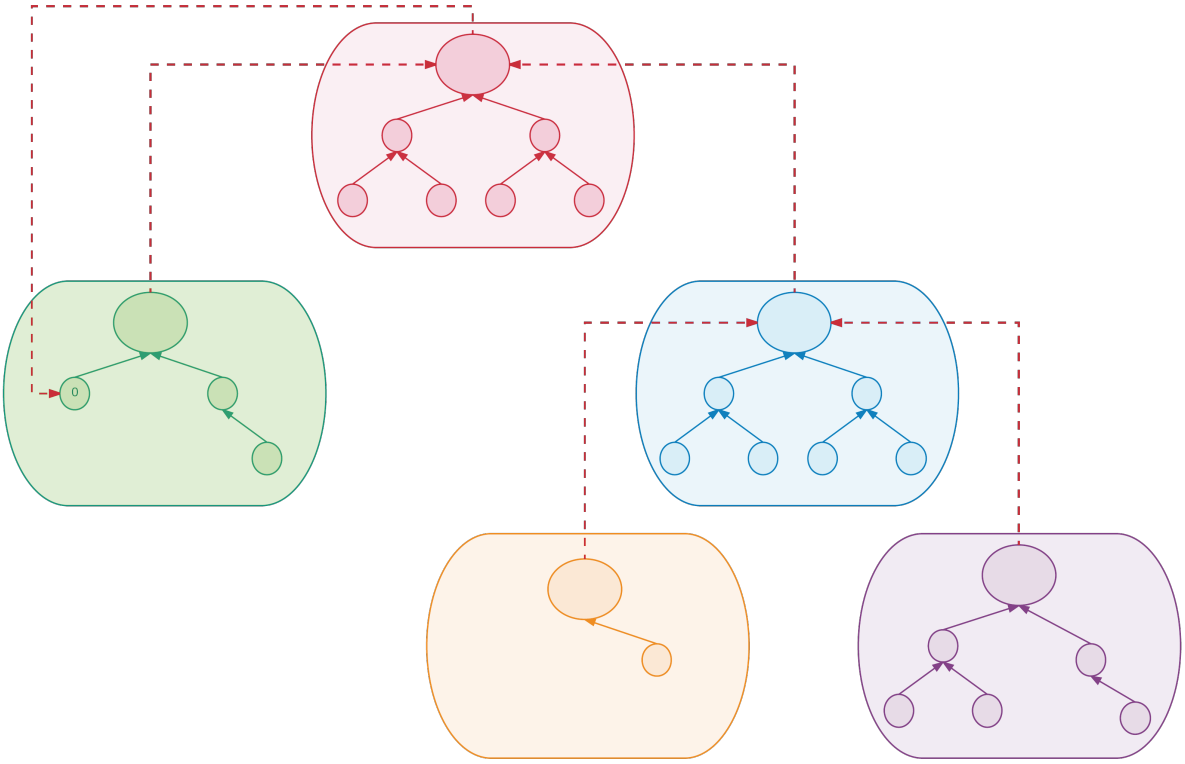

Figure 4.11: Flow of monitoring data in example topology

In the network topology there are four types of specialized nodes:

- **Leaf nodes**: These are the nodes at the bottom of the *Local Tree*. They only gather data from themselves and send it to their parent node. This functionality's flow of execution is explained in figure 4.19.
- **Regular nodes**: These are the intermediate nodes of the *Local Tree*. That is, the nodes who have at least one child node from which to receive monitoring data and who also have a parent node to which send its own data and the one received from the child or children nodes. This flow of execution is described in figure 4.21.
- **Root local nodes**: These are the ones which are the root of their *Local Tree*. This means that they gather data from themselves, from their children nodes, not only in the *Local Tree*, but also from their children nodes in the *Global Tree* if they have any, and they send that data to their parent node in the *Global Tree*. The flow of execution of this node is detailed in figure 4.20.

- **Zero nodes**: There is only going to be only one instance of these kind of node in the application, as this node represents the process whose rank is 0. This node can be part of a *Local Tree*, but cannot be part of the Global one. This node would send its own data to its parent node in the *Local Tree* if it has, it would receive data from its children in the *Local Tree* if any, and also it would receive data from the root of the *Global Tree*, for later showing it in the screen. The flow of execution of this node is detailed in figure **??**.

These nodes would read every second the average time taken for the iteration to compute. This average time is calculated by subtracting the end time of the loop to the start time of it, then, this difference is subtracted to the average time, which is set to 0 in the first iteration, and then it is divided by the average.

If this product is higher than 0.05, then the average time is updating by multiplying it by 0.8, and adding the difference calculated multiplied by 0.2. A piece of pseudocode of this process is provided for clarification. Also, this flow of execution is detailed in figures 4.17 and 4.15 respectively.

```
difference = |loop start time - loop end time|

if((|average - difference|)/average > 0.05) {
    average = 0.8 * average + 0.2 * difference
}
```

Listing 4.4: Average iteration time calculation

The communication among these nodes is performed using the capabilities of the EVpath library. EVpath allows the creation of receiving and sending processes whose functionality will be used in our nodes.

For the receiving nodes, a string representation of the nodes, called *Contact List* is generated. This string contains the essential information about how that node can be contacted by the nodes who are supposed to send data to it. It has the following attributes:

- Stone ID: This is an integer number which represents the EVpath "stone" attribute.
- Address Information: This is the ID of the process running an EVpath receiving side.
- Filtering function: This is a 64 based encoded string which represents a C function to be run by the sending processes in order to filter the data to send.

A Contact list has the following syntax:

Figure 4.12: Example of an EVpath contact list

Note that the three attributes are separated by a colon (:) which is used as a character to split the contact list in the three attributes.

These Contact lists generated in the receiving nodes have to be transmitted to the sending ones in order to know where to connect to for establishing the communication. This is done through a pair of MPI functions called *MPI_Send*, in the receiving side, and *MPI_Recv* in the sending side. With this information available through the nodes, the communication protocol of the different specialized nodes can be started.

The spreading of the Contact lists through the nodes in the overlay follows the opposite direction to the messages sent in the communication mechanism of the monitoring data, as shown below:

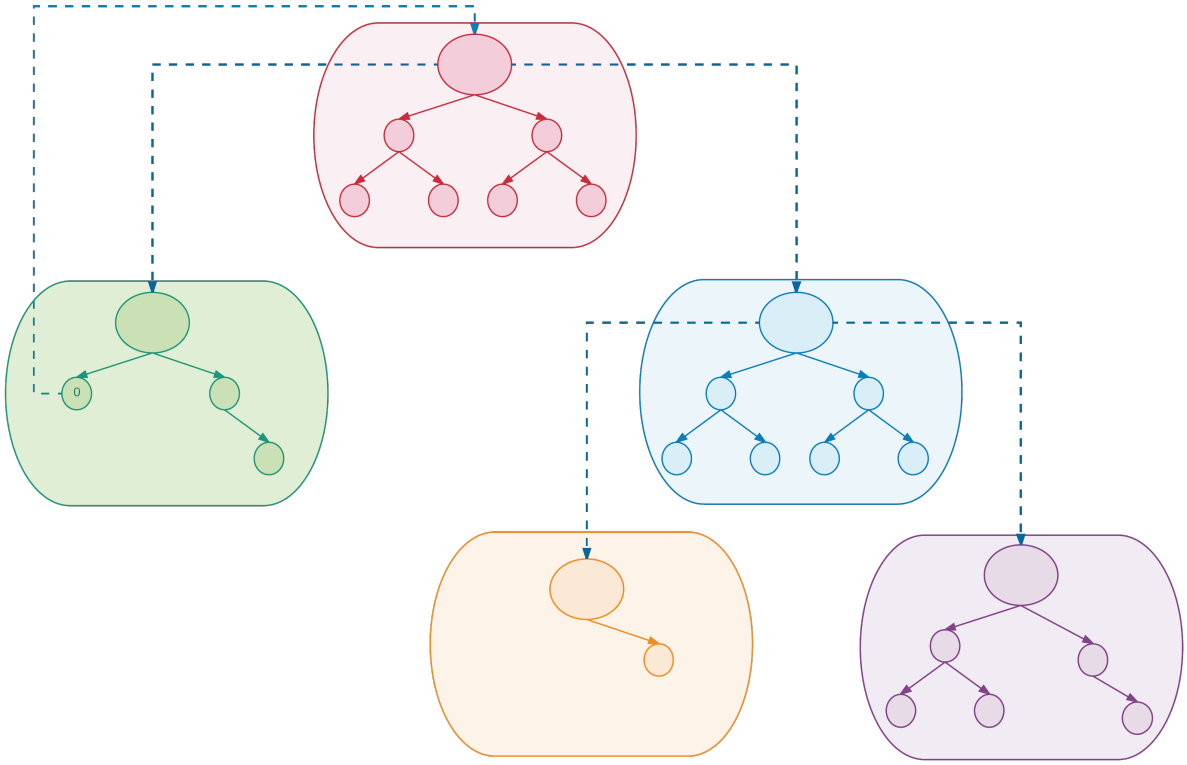

Figure 4.13: Flow of transmission of Contact lists generated in example topology

The execution flow configuration of the node topology can be seen in figure 4.16.

Balanced Binary Search Tree

In order to create the Global and Local trees, as stated before, a Balanced Binary Search Tree data structure is employed.

This data structure is created in a recursive way by calling a function called *bst* whose arguments are an array of values, in this case *Comm* structures, the first position of the array, which is usually 0, and the last position of the array.

Then, a node is created for the element in the middle of the array, it is inserted as the root of the tree and the function is called again for creating a subtree with the elements on the left of the middle element of the array, and another one for the elements on the right of the middle element of the array. This process is repeated in a recursive way until there are no more elements on the array to be inserted in the tree. The flow of execution of this is described in detail in figure 4.24.

The usage of a binary search tree structure to create the topology has been chosen over other data structures like singly-linked lists for complexity reasons. This selection allows the system to create the network topology in two data structures for each node whose relevant complexities are as follows:

Time Complexity								Space Complexity
Average				Worst				Worst
Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Table 4.39: Complexity of binary tree

In the table 4.39 we can observe that on average, the operations of access, search, insertion and deletion of a binary search tree are O(log(n)), which is fairly similar to O(1), which means direct access and is the best case for complexity in any data structure. The only parameter which would perform worse is on space complexity, as all binary trees use an amount of memory which grows linearly with the number of nodes, but since most data structures behave in the same way, this is not a great inconvenient.

The trees, as stated, are deleted upon the communication mechanism is started. In order to do that, the same tree is walked recursively deleting every node. This is explained in detailed in figure 4.25.

4.4.4 Flow diagrams

In this section flow diagrams which represent how the different functionalities of the project developed, described in the previous section, are presented. First the ones stating how the monitorization functionalities, which create the network topology and update the data to monitor, and last the ones in charge of the communication and filter of that data:

Figure 4.14 represents how the functionality to start the monitorization mechanism works. The functionality is initialized, the variables *t1*, *t2*, *dt* and *avg_dt*, which are the ones used to compute the time it takes to execute an iteration of the main loop of the application, are initialized to 0. Then the variable *running* is set to 1, in order for the monitorization mechanism to know the application to monitor is running. Finally, the thread which executes the monitorization mechanism is launched. This thread functionality is represented in figure 4.14.

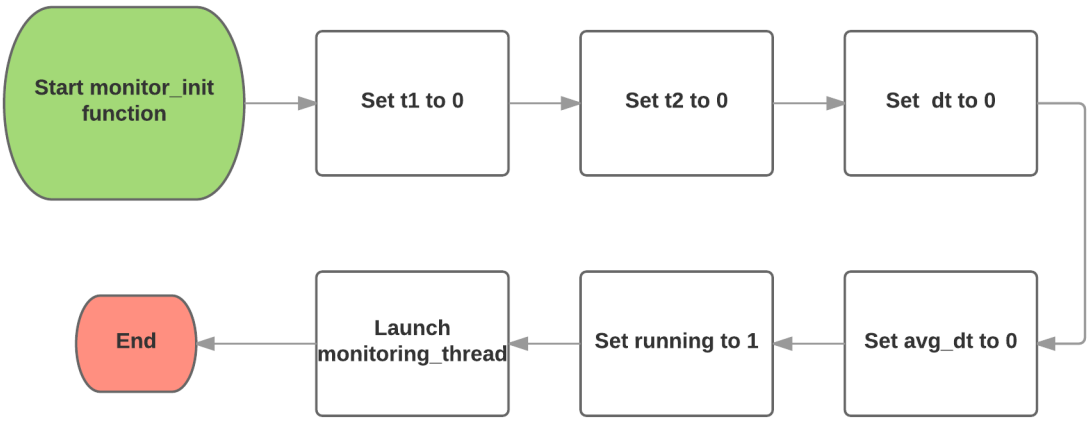

Figure 4.14: Monitorization start functionality diagram

In figure 4.15 the procedure to get the time at which the main loop ends executing, and calculating the average time it takes for the iteration to execute, is described. The functionality starts getting the end time of the iteration and it calculates the difference between it and the start time of the iteration. If the difference is 5% larger is smaller tan the average time, it gets updated, if not, nothing is done.

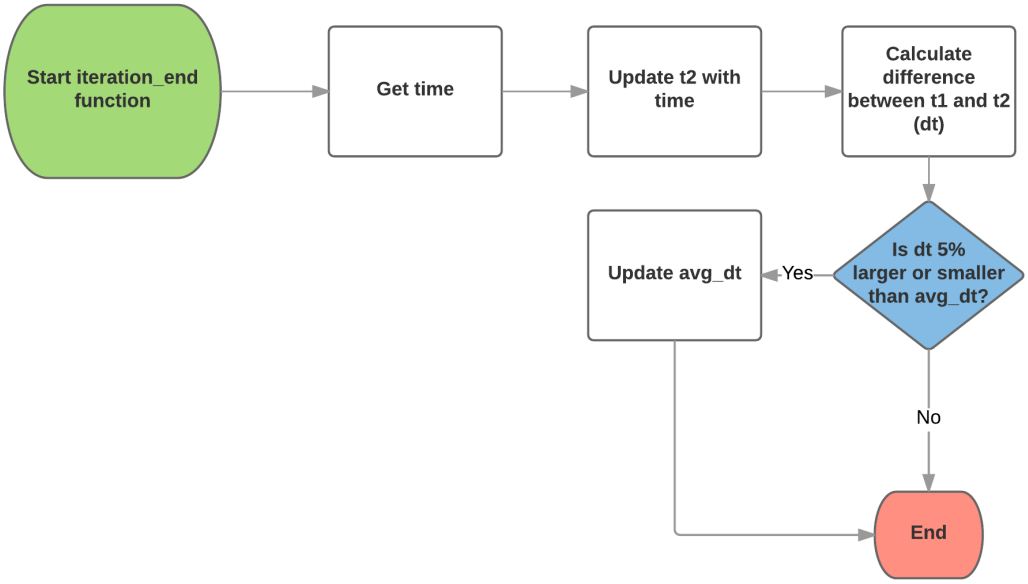

Figure 4.15: Iteration end functionality diagram

In figure 4.16, the flow of execution of the functionality of the creation of the topology is explained. First, the information from all processes is aggregated by each node, and with that information, the Local Tree for the processes residing in the same node is created. Then, with the information for these Local Trees, the communication information for them is established. Each process aggregates the relevant information from all the nodes of the system in order to obtain the relevant information about the communication inter nodes. Then the root nodes of all the local trees are used for creating the Global Tree and configure all the inter nodal communications. Then the root of the Global Tree is set to send to the node whose ID is zero. Finally each node checks which kind of node it is and executes its corresponding functionality.

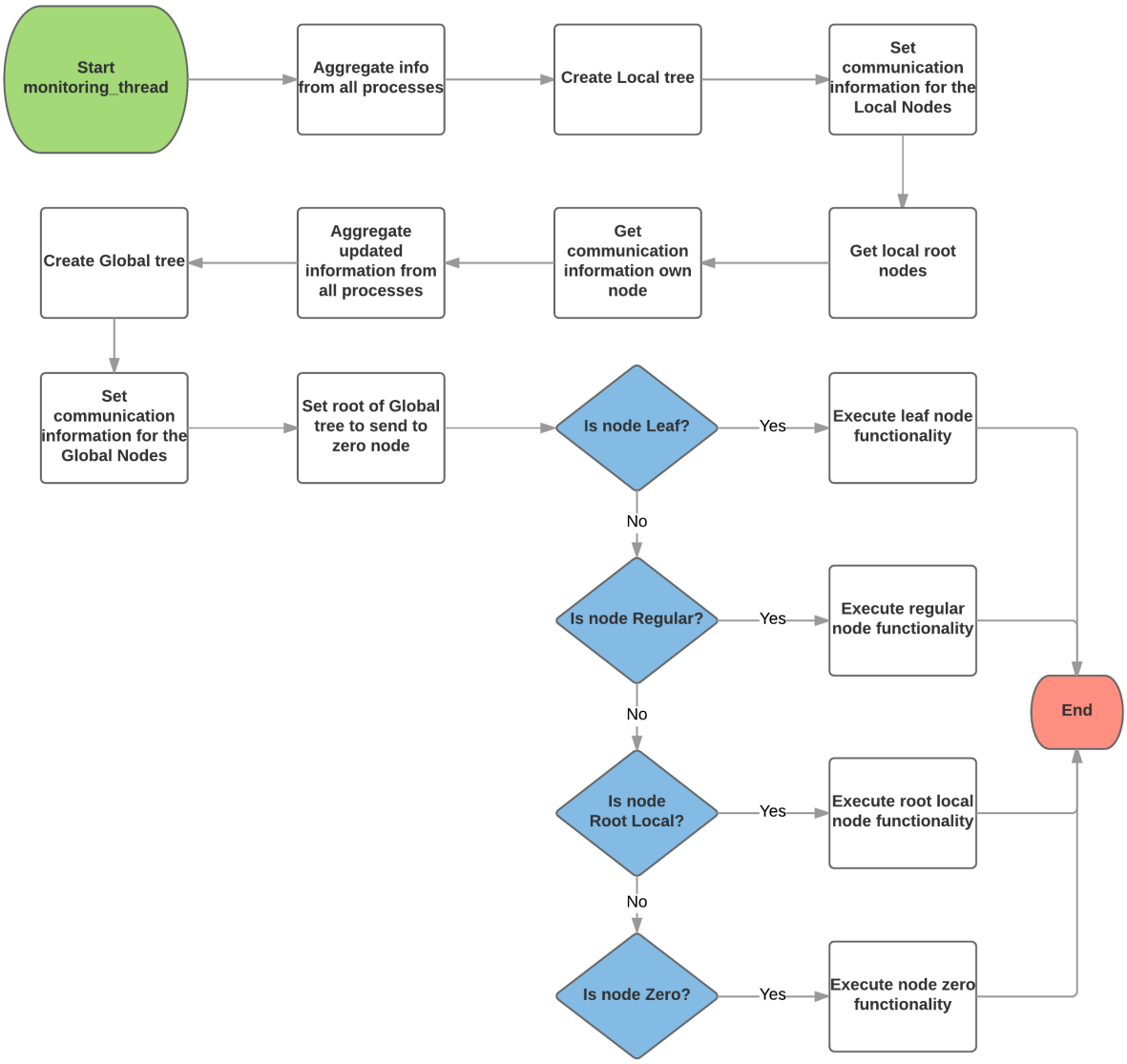

Figure 4.16: Monitorization thread functionality diagram

In figure 4.17, the flow of execution of the function to get the time at the start of the main loop of the application is explained. Once the functionality starts, it gets the time at that moment and stores it in the global variable *t2*.

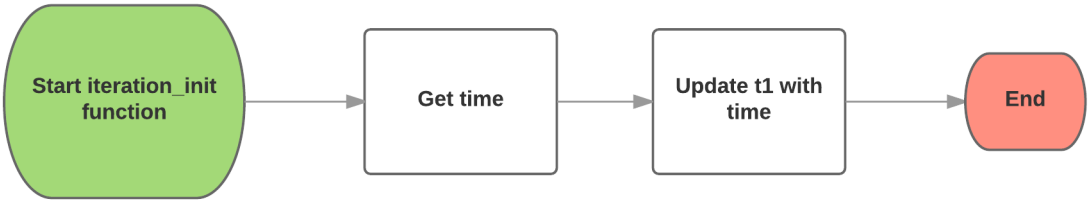

Figure 4.17: Iteration init functionality diagram

The figure 4.18 shows how the functionality of stopping the monitorization mechanism works. First, it sets the variable *running* to 0, which will make all the launched threads to end. Then it waits for them and when they are finished the function ends.

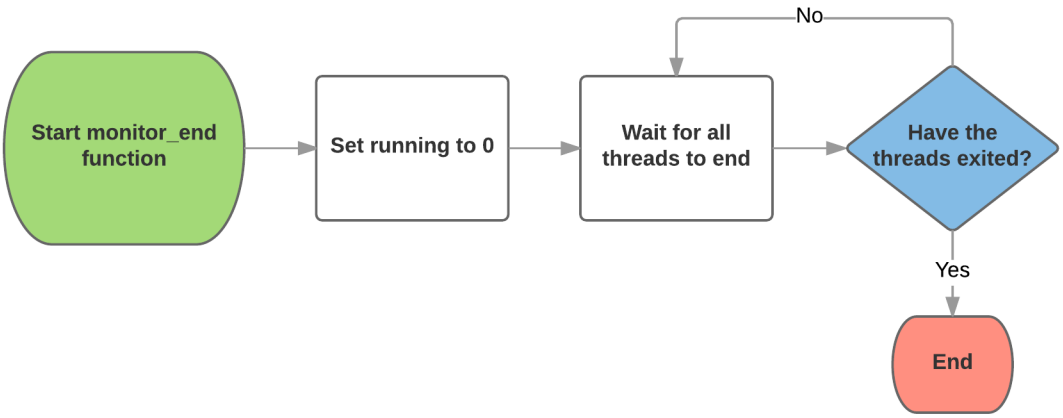

Figure 4.18: Monitorization end functionality diagram

The figure 4.19 shows how the functionality of stopping a specialized leaf node works. First, it waits for the contact list from its parent. When received, the filtering function is obtained from it, and the connection is established with the parent. If the variable *running* is 1 and the filter conditions stated by the filtering function is met, the data is sent to the parent. Once the variable *running* is 0, the communication is finished, the process exits and the functionality ends.

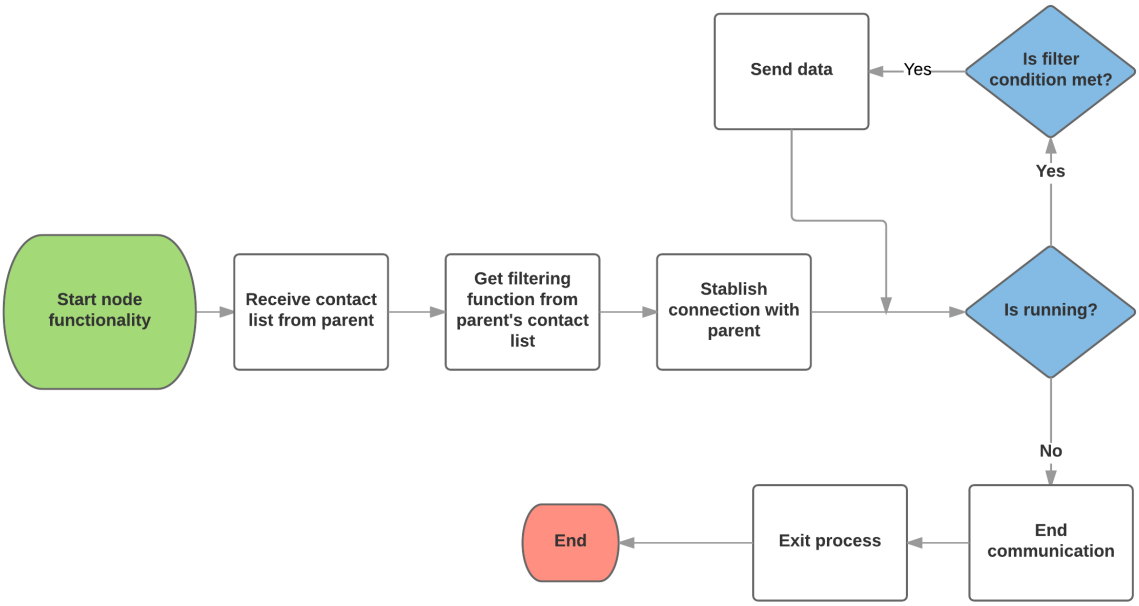

Figure 4.19: Leaf node functionality diagram

The figure 4.20 shows how the functionality of the intermediate nodes of a local tree is executed. First, they wait for the contact list to be received from their parent. Once received, the filtering function is obtained from it. They create their own contact list to send to their children, if any.

A thread that will read the monitorization data is launched with the node's own contact list as argument. The thread gets the filtering function from the contact list and establish a connection with themselves. If the variable *running* is 1 and the filter condition of the filtering function is met, the data is sent repeatedly until the *running* variable is 0, which would end the communication and exit the thread.

After the thread is launched, the node sends its contact list to their left and right local and global children, if any. If the variable *running* is 1, it waits for messages from their children and themselves to be resent to their parent until the *running* variable is 0, which would end the communication and exit the process. If any data is received, they establish a connection with their parent, and if the filter condition is met, the data is sent to their parent.

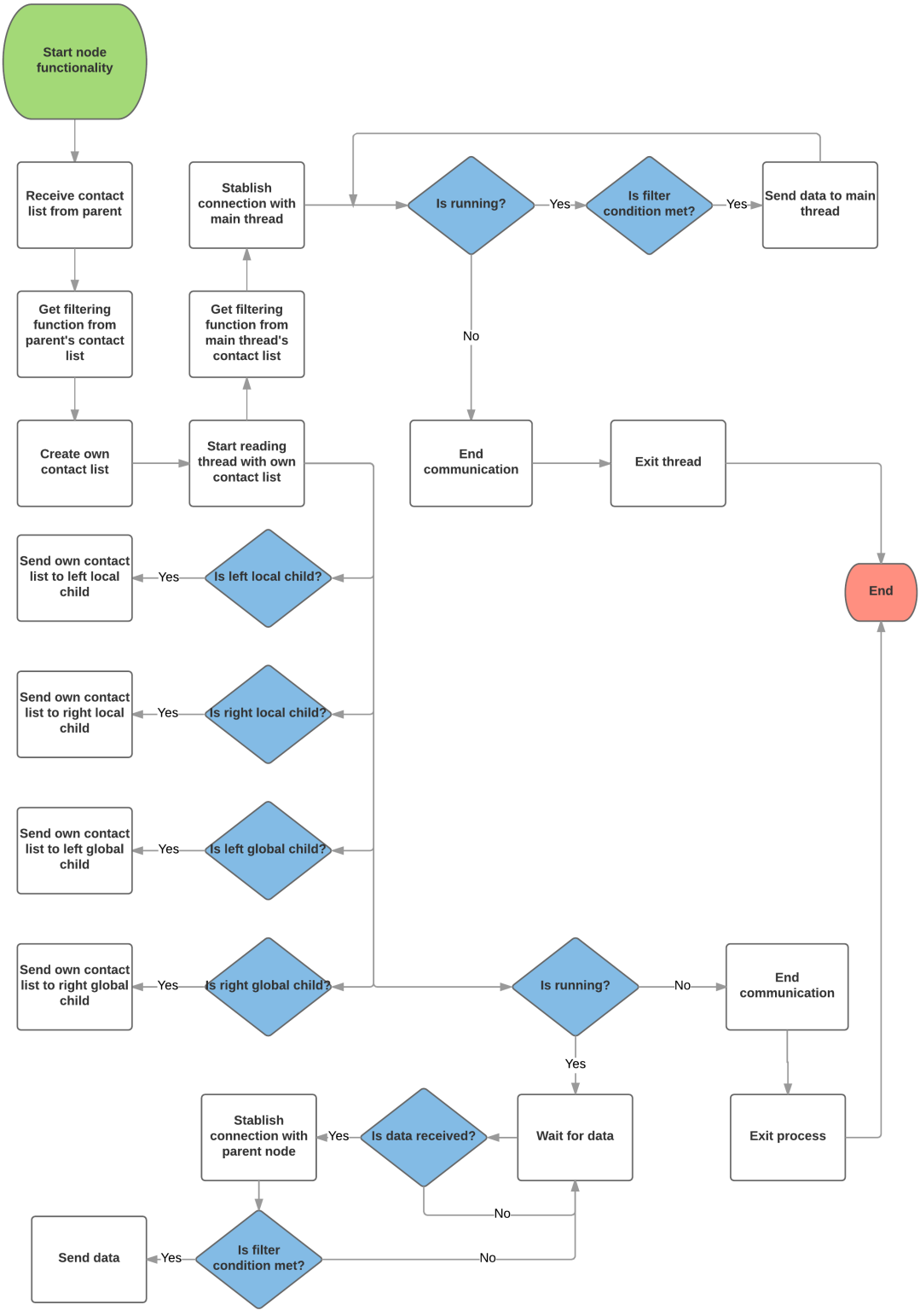

Figure 4.20: Root local node functionality diagram

The figure 4.21 shows how the functionality of the intermediate nodes of a local tree is executed. First, they wait for the contact list to be received from their parent. Once received, the filtering function is obtained from it. They create their own contact list to send to their children, if any.

A thread that will read the monitorization data is launched with the node's own contact list as argument. The thread gets the filtering function from the contact list and establish a connection with themselves. If the variable *running* is 1 and the filter condition of the filtering function is met, the data is sent repeatedly until the *running* variable is 0, which would end the communication and exit the thread.

After the thread is launched, the node sends its contact list to their left and right children, if any. If the variable *running* is 1, it waits for messages from their children and themselves to be resent to their parent until the *running* variable is 0, which would end the communication and exit the process. If any data is received, they establish a connection with their parent, and if the filter condition is met, the data is sent to their parent.

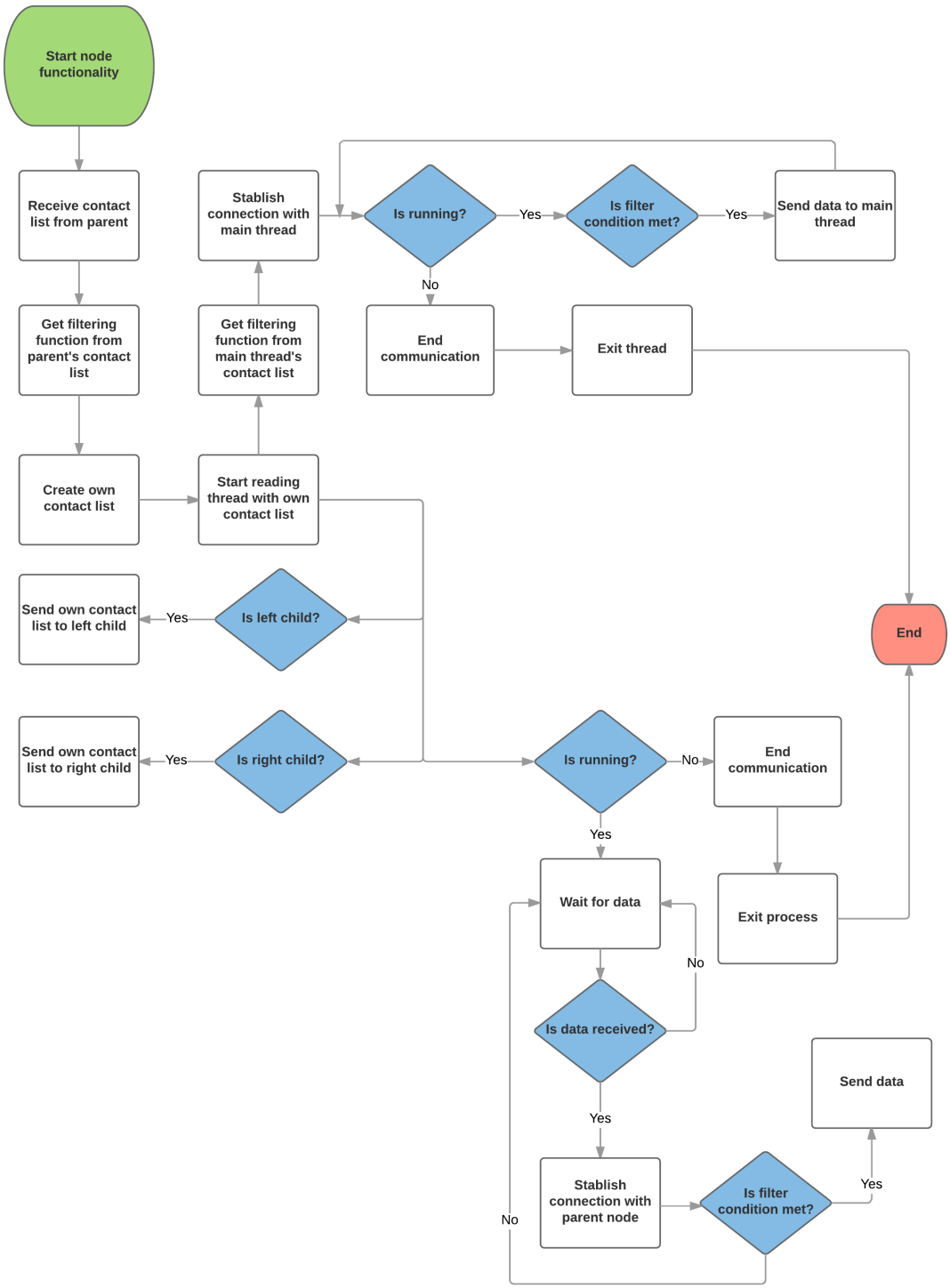

Figure 4.21: Regular node functionality diagram

In figures 4.22 and 4.23 the flow of execution of the node whose ID is 0 is stated. First, if there is a global right child, that means there is at least one node in the Global Tree, so a thread is launched to listen for this process messages. In this thread, a contact list is generated, and sent to the right global child. If the variable *running* is 1, it waits for messages from the root of the Global Tree to be resent to their parent until the *running* variable is 0, which would end the communication and exit the thread. If any data is received, it is shown on the screen.

Second, it checks if there is a parent node, if so, it wait for the contact list from the parent and get the filtering function from it. Then it would create its own contact list to launch a thread which would get the filtering function from the contact list and establish a connection with itself. If the variable *running* is 1 and the filter condition of the filtering function is met, the data is sent repeatedly until the *running* variable is 0, which would end the communication and exit the thread.

Then it would send its generated contact list to its local children, if any, and if the variable *running* is 1, it waits for messages from their children and itself to be resent to its parent until the *running* variable is 0, which would end the communication and exit the process. If any data is received, it establish a connection with its parent, and if the filter condition is met, the data is sent to their parent.

In case there is no parent node, it would create its own contact list to launch a thread which would get the filtering function from the contact list and establish a connection with itself. If the variable *running* is 1 and the filter condition of the filtering function is met, the data is sent repeatedly until the *running* variable is 0, which would end the communication and exit the thread.

Then it would send its generated contact list to its local children, if any, and if the variable *running* is 1, it waits for messages from their children and itself until the *running* variable is 0, which would end the communication and exit the process. If any data is received, it is shown on the screen.

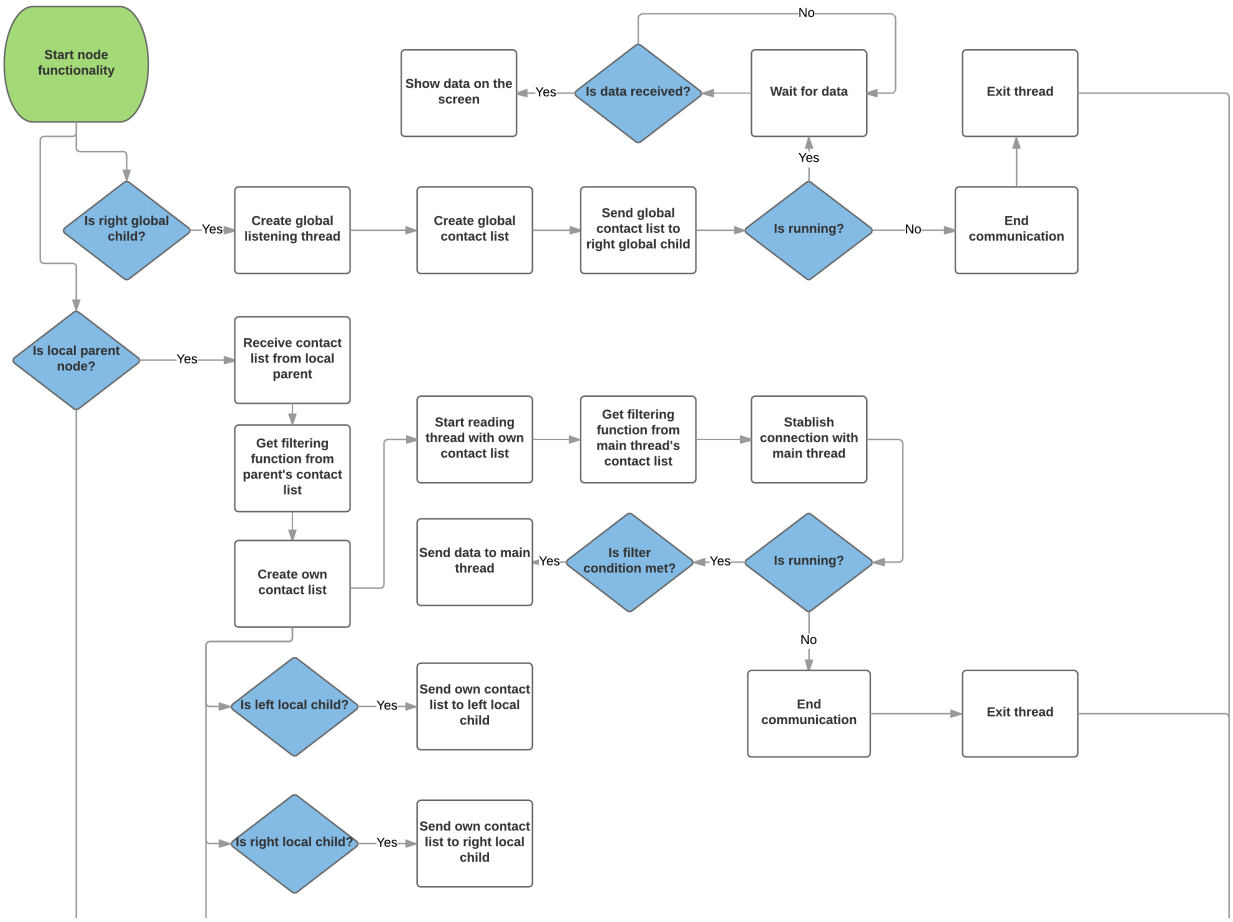

Figure 4.22: Node zero functionality diagram

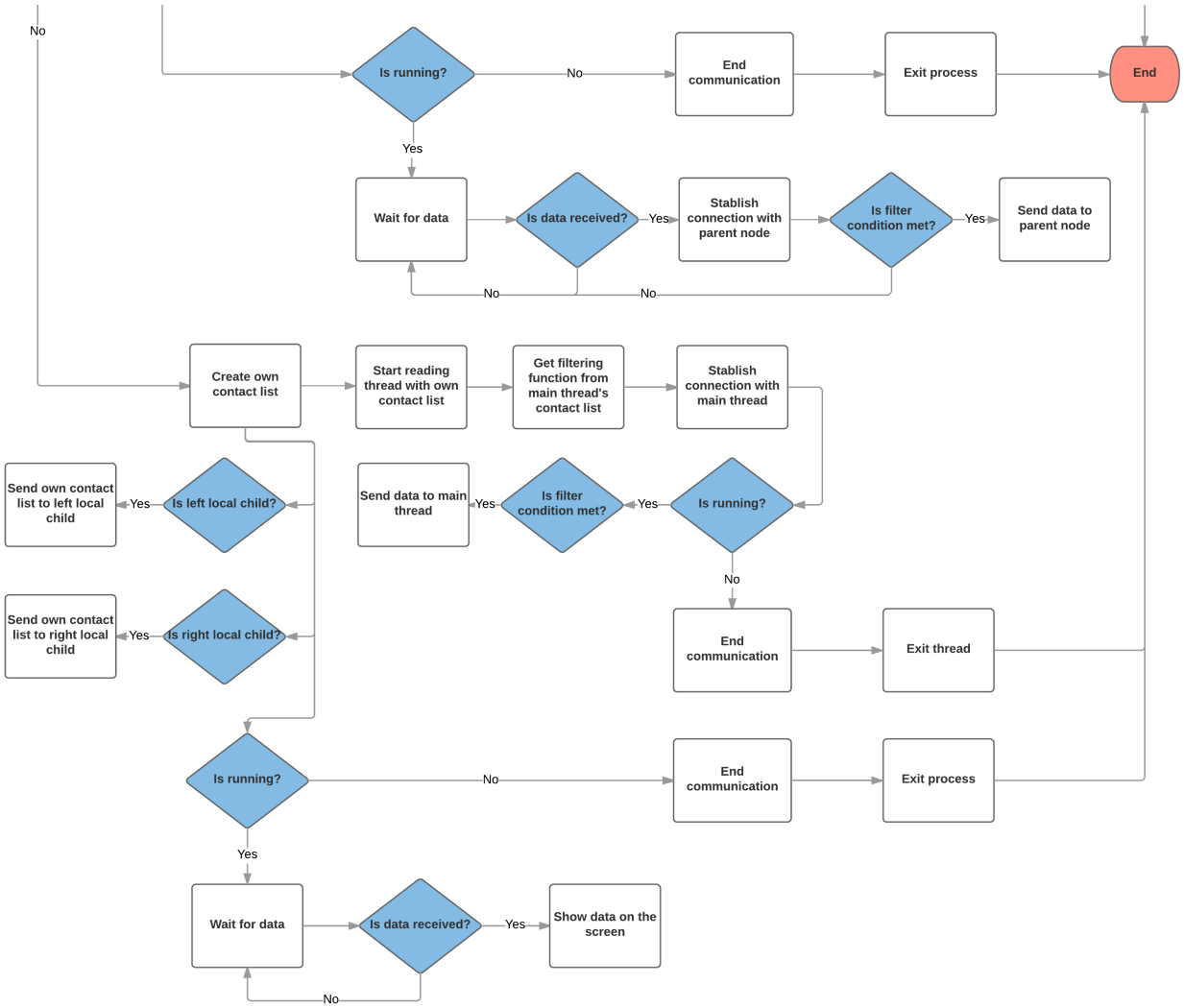

Figure 4.23: Node zero functionality diagram

In figure 4.24 the flow of execution of the creation of a balanced binary search tree is explained. This functionality takes as parameters, an array of elements to be introduced in the tree, the first value of the array and the last. First, it checks if the first value given is greater than the last. If so, the functionality ends. Then it creates a node with the element in the middle of the array and sets its left child to a recursive call with the array, the first value of the array, and the element to the left of the middle element of the array as parameters. For the right child, it performs the same action but with the element at the right of the element in the middle on the array and the last element of the array as parameters.

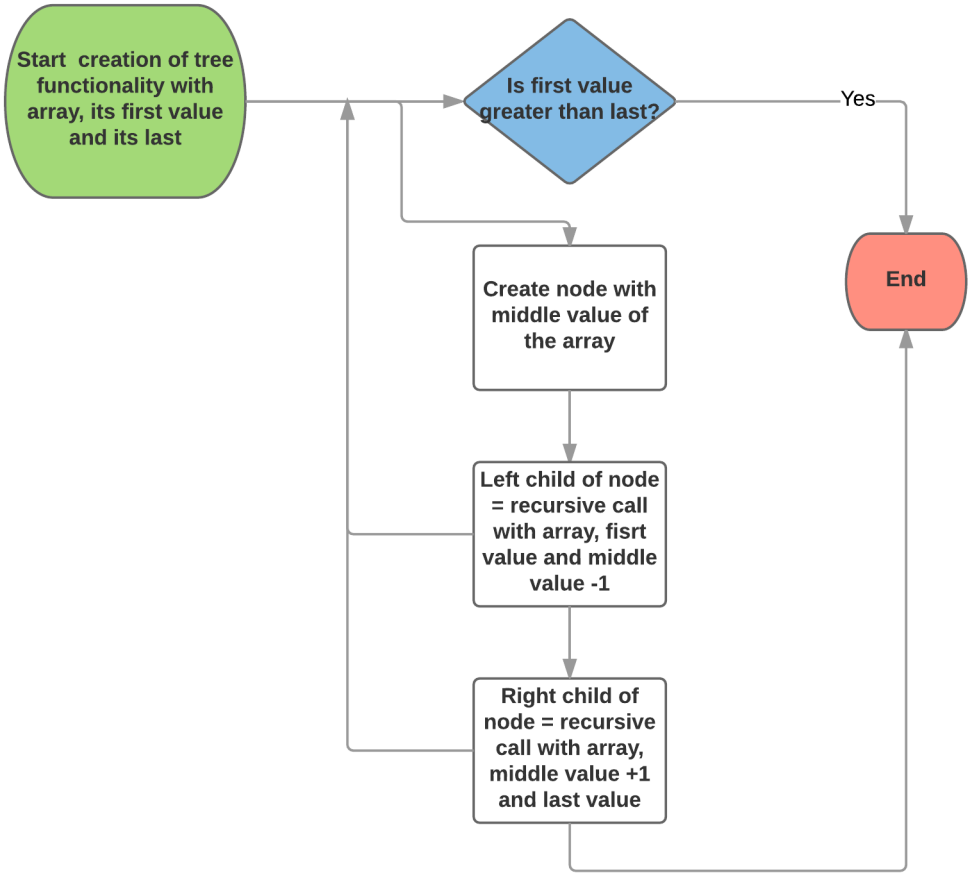

Figure 4.24: Creation of balanced binary search tree diagram

In figure 4.25 the flow of execution of the functionality which deletes a tree is stated. It receives as parameter the root node of the tree. First it checks if the node passed as parameter is null, if so, the execution ends. Then it performs a recursive call with the left child of the node, and another with the right.

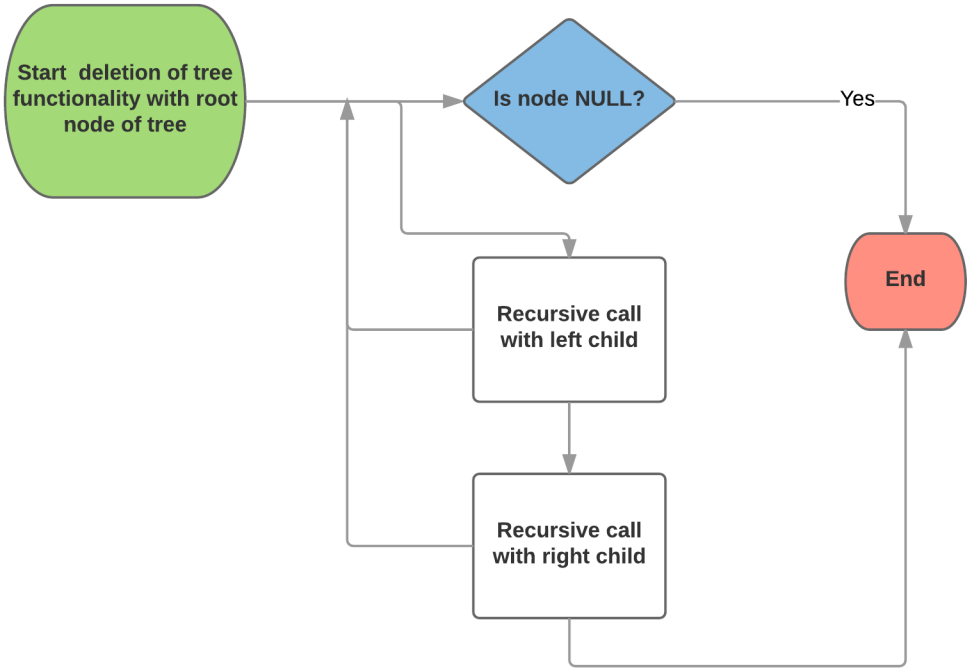

Figure 4.25: Deletion of balanced binary search tree diagram

In figure 4.26 the flow of execution of the configuration of the network with the Global Tree is explained. It receives the root node of the tree as parameter. First it checks if the node has a parent, if so, it sets the ID of the parent as ID of the node to send data to. If the node has a left child, its ID is set as one of the IDs from which it receives data, then it performs a recursive call with the left child as parameter. Also, if the node has a right child, its ID is set as one of the IDs from which it receives data, then it performs a recursive call with the right child as parameter.

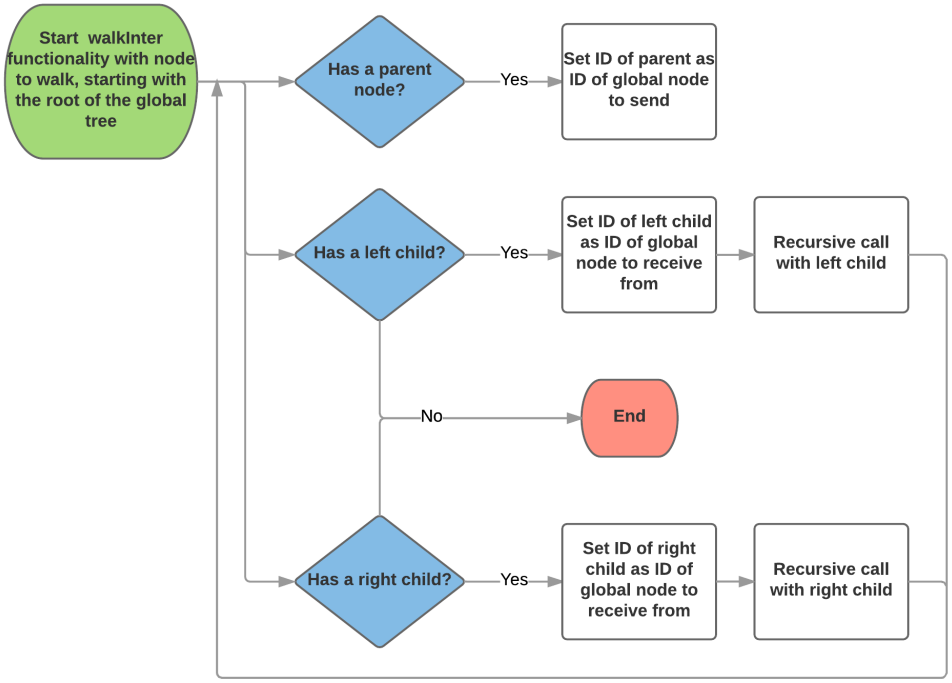

Figure 4.26: Configuration of global network topology diagram

In figure 4.27 the flow of execution of the configuration of the network with the Local Tree is explained. It receives the root node of the tree as parameter. First it checks if the node has a parent, if so, it sets the ID of the parent as ID of the node to send data to. If the node has a left child, its ID is set as one of the IDs from which it receives data, then it performs a recursive call with the left child as parameter. Also, if the node has a right child, its ID is set as one of the IDs from which it receives data, then it performs a recursive call with the right child as parameter.

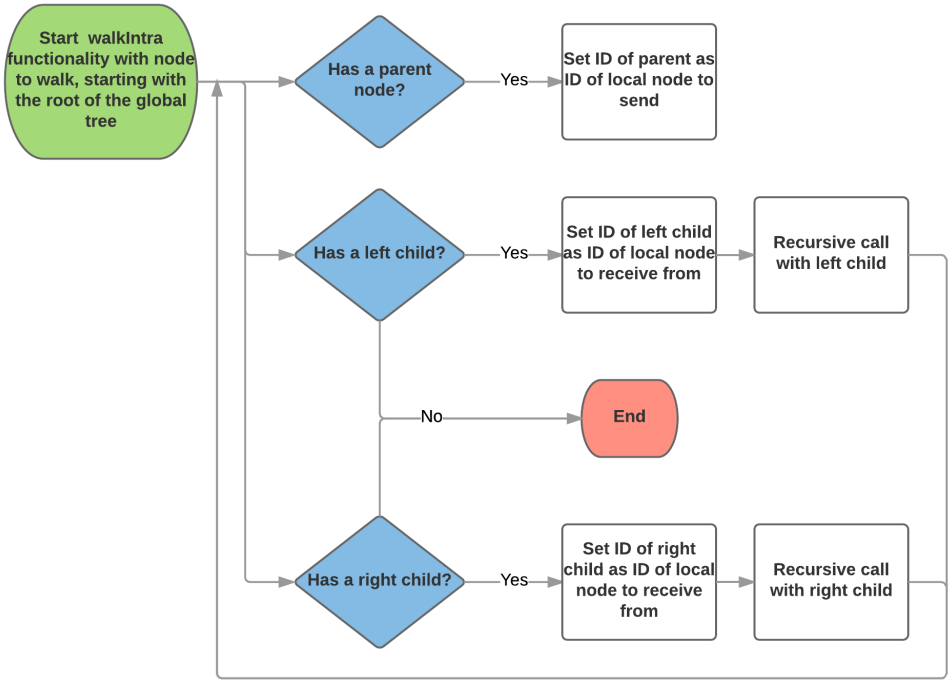

Figure 4.27: Configuration of local network topology diagram

4.5 Project planification

The planification of the project has been divided in six main phases:

- System requirements:
  - Functionality definition
  - Use cases definition
  - Requirements definitions
  - Budget definition
- Analysis
  - EVpath analysis
  - MPI analysis
  - Requirements analysis
  - Use cases analysis
- Design
  - System design
- Implementation
  - Initial simple implementation
  - Balanced binary search tree implementation
  - Tree shaped topology configuration
  - Integration with existing software
- Testing
  - Testing in hardware environment
- Documentation
  - Report documentation
  - Presentation

The project started in the first day of February and finished the 22nd of September, being the total duration of it around the 8 months. It must be taken into account that the development of the different tasks have been concurrent with other tasks non related to this project, like work, exams or university projects, which have delayed considerably some of them.

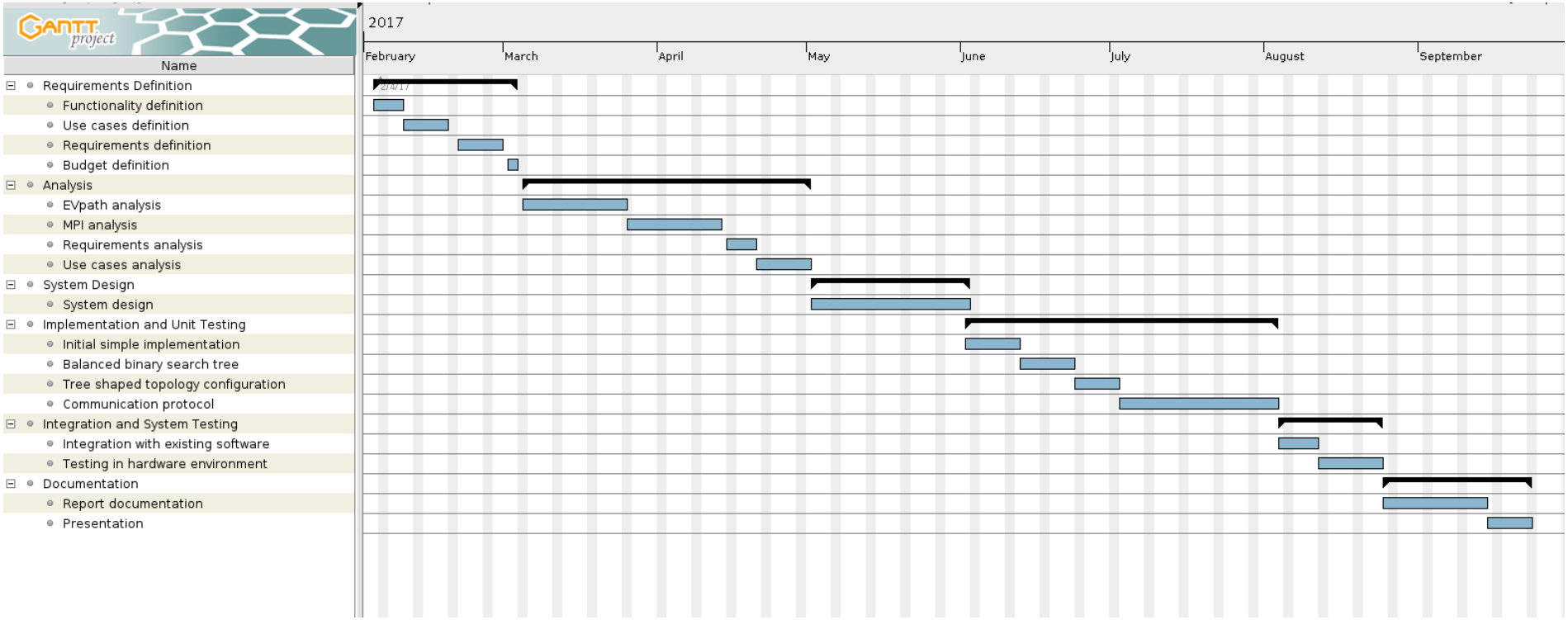

Figure 4.28: Project Gantt Diagram. Dates are in MM/DD/YYYY format.

4.6 Legal Framework

In this section, the different legal regulations that apply to the project are presented.

4.6.1 Applicable legislation

From the point of view of the developed system itself, it only manages monitorization data that itself does not represent any personal, confidential or sensitive information, but the parallel applications that the system monitors can work with personal data. For that, in Spain, the applications to be monitored must comply with the organic law 15/1999 on the protection of personal data (LOPD), whose purpose is to guarantee and protect the public liberties and fundamental rights of the physical people in the treatment of the personal data. [20]

4.6.2 Technical standards and intellectual property

First, the licenses in place in the external libraries that the system uses must be analyzed.

The MPI implementation used, MPICH, is distributed under a BSD-Like license, which grants the use, repreduction, preparation of derivative works and redistribution to others. [21]

The EVpath library, is distributed under the New BSD License or 3-Clause BSD license which grants redistribution and use in source of binary forms with or without modifications given that the following conditions are met: [22]

- The redistributions of the code must retain the copyright notice, the list of conditions and the disclaimer stated in the license.
- The redistributions in binary form must reproduce the copyright notice, the list of conditions and the disclaimer stated in the license in the documentation and/or materials provided with the distribution.
- Neither the name of the copyright holder, nor the names of the contributors may be used to endorse or promote products derived from the licensed software without prior written permission.

  The developed software is licensed under the Mozilla Public License (MPL) which is compatible with both the two licenses cited above.

4.7 Socio-economic environment

4.7.1 Budget

In this section the costs of development of the total project are broken down in order to estimate the total budget for it.

In the following table, the estimated costs for human resources, as direct costs, are presented. It is added to the base salary the Social Security costs [23], which is 28.3% of the salary. The base salary has been obtained from the BOE (Boletín Oficial del Estado) for the 18th of January of 2017, Number 15. [24]

Position	Monthly base salary	Number of months	Final cost	Final cost with SS
Programmer	1208.40€	2.37	2859.88€	3669.22€
Tester	1208.40€	0.47	563.92€	723.50€
Designer	1208.40€	1.07	1288.96€	1653.74€
Analyst	1687.02€	1.93	3261,57€	4184.60€
Project Director	1253.16€	1.3	2708€	2143.74€
TOTAL				12374.8€

Table 4.40: Estimated staff associated direct costs.

The next table the costs related to equipment used are listed. These are composed of the usage of a laptop for development and the nodes of the Tucán cluster as they have been used in the development of the project. This has caused them a degradation whose cost is estimated below.

Concept	Unitary cost	Units	Amortization per month	Chargeable cost
Development laptop	1100 €	1	70.50 €	564 €
Node compute-8-1	1336.19 €	1	90.75 €	726 €
Node compute-9-1	5264.97 €	1	175.25 €	1402 €
Node compute-9-2	5087.99 €	1	125 €	1000 €
Node compute-9-3	6694.75 €	1	275 €	2200 €
TOTAL				5892 €

Table 4.41: Estimated equipment direct cost for a use of 6 months.

In the following table the costs indirectly related to the project are listed. As mentioned above, the project runs in a cluster in the University which needs maintenance, a room, power and internet connection. These costs are estimated below.

Description	Monthly cost	Amortization per month	Total cost
Cluster room rent	1100 €	110 €	880 €
Energy costs and CPD maintenance	600.35 €	60.35 €	482,8 €
Internet connection	60.33 €	5.33 €	42,64 €
TOTAL			1405.44 €

Table 4.42: Indirect costs.

Aggregating all these costs, we obtain a total cost of 19672.24 € without VAT. Adding taxes, which are 21%, the final cost is **24623.79 €**. Assuming a profit of 10% of the total cost of the project, it would be of 2462.38 €.

4.7.2 Socio-economic impact

In this section the socio-economic impact of the system developed is presented.

The social benefits appointed by this project would be directed to the usage of parallel applications in high-performance computing systems. These applications, that require the usage of inter nodal communication, would benefit from a system like the one developed as it provides an scalable way for the communication to be executed.

For the economic impact, the system developed has been developed under a Mozilla Public License (MPL) which is compatible with all the other libraries used in the development of the project, as they all use the BSD license. [25] The MPL license grants the right to use the code as commercial software, to use, analyze, modify and distribute it and to combine it with other licenses without holding the original developer liable. [26]

Since it is a free and open software, the economic impact would be zero, but once applied to a production environment, in parallel applications, it could help to save costs in terms of power consumption, as avoiding computational bottlenecks can cut these kind of costs which also, indirectly would have a positive environmental impact. Also, there could be benefits by providing user and developer support for the system the way RedHat does with their products.

Chapter 5

Evaluation

In this chapter the system is evaluated and specified how this has been done. First the hardware platform in which the tests have been performed is described. Second, the different case studies run in order to know if the system works are listed. Third, the trazability matrix that traces which requirements are covered by which case studies is presented. Lastly, the different performance tests designed are exposed and the results shown and compared among them.

5.1 Description of the hardware platform

The hardware platform in which the tests have been performed to the system is the Tucán cluster of the ARCOS research group in the Universidad Carlos III de Madrid[TODO], which, as stated previously is a distributed and shared memory system. The different nodes used in order to pass the tests have different characteristics, which are cited below.

- **Node compute-8-1**: This node presents a CPU model Intel(R) Xeon(R) CPU E5-2620 0 whose frequency is of 2.00GHz and have a total number of 6 cores. This processor has been designed for server equipment. It counts with hyperthreading technology which adds a logical core to each physical one, making the system believe it counts with 12 cores. Also, it has a cache memory of 15 MB. In terms of main memory it has a RAM memory of 78 GB. In terms of storage it counts with a 1 TB TOSHIBA MK1002TS hard drive disk and a 512 GB Samsung 850 solid state drive.
- **Node compute-9-1**: This node presents a CPU model Intel(R) Xeon(R) CPU E5-2630 v3 whose frequency is of 2.40GHz and have a total number of 8 cores. This processor has been designed for server equipment. It counts

with hyperthreading technology which adds a logical core to each physical one, making the system believe it counts with 16 cores. Also, it has a cache memory of 20 MB. In terms of main memory it has a RAM memory of 252 GB. In terms of storage it counts with two 1 TB WDC WD1003FZEX-0 hard drive disks and a 512 GB Samsung 850 solid state drive. Also, it counts with two GPU units GeForce GTX TITAN Black.

- **Node compute-9-2**: This node presents a CPU model Intel(R) Xeon(R) CPU E5-2630 v3 whose frequency is of 2.40GHz and have a total number of 8 cores. This processor has been designed for server equipment. It counts with hyperthreading technology which adds a logical core to each physical one, making the system believe it counts with 16 cores. Also, it has a cache memory of 20 MB. In terms of main memory it has a RAM memory of 378 GB. In terms of storage it counts with two hard drive disks SMC3108, one with a capacity of 557.9 GB and the other one with 2.7 TB. It also counts with one 1 TB ST1000DM003-1CH1 solid state drive. Also, it counts with two GPU units GeForce GTX 680.

- **Node compute-9-3**: This node presents a CPU model Intel(R) Xeon(R) CPU E5-2630 v3 whose frequency is of 2.40GHz and have a total number of 8 cores. This processor has been designed for server equipment. It counts with hyperthreading technology which adds a logical core to each physical one, making the system believe it counts with 16 cores. Also, it has a cache memory of 20 MB. In terms of main memory it has a RAM memory of 252 GB. In terms of storage it counts with two 1TB hard drive disks WDC WD1003FBYZ-0 and one 512 GB Samsung 850 solid state drive. Also, it counts with two GPU units, a Tesla K40c and a GeForce GT 610.

5.2 Case studies to evaluate that system works

In this section the different case studies designed in order to evaluate that the system work are presented.

Each case is represented with a table like Table 5.1

Case study	
ID	CS-XX
Objective	Objective of the case study
Preconditions	Conditions to be met before the case study
Sequence	Description of the execution
Postconditions	Conditions met after the case study
Result	Verified or Not verified
Trazability	Requirements analized

Table 5.1: Example of case study table

The attributes of a case study are described below:

- ID: Unique identifier of the case study. It is formed by the use case's code CS followed by a "-" and two digits.
- Objective: Objective to accomplish in the case study.
- Preconditions: Conditions to be met before the execution of the case study.
- Sequence: Steps to be run by the case study.
- Postconditions: Conditions to be met after the execution of the case study.
- Result: Result obtained after the execution of the case study. The result can be Verified or Not verified.
- Trazability: Relation with the requirements of the system.

Following up, the case studies are listed.

Case study	
ID	CS-01
Objective	Verify that the system executes the stated processes in the required nodes of the cluster
Preconditions	The system must be executed in a cluster with a rankfile specified
Sequence	Run the specified processes in the specified nodes of the cluster
Postconditions	The system would execute the desired processes in the speficied nodes as required
Result	Verified
Trazability	NFR-03, NFR-04, NFR-08, NFR-09 and NFR-10

Table 5.2: Case study CS-01

Case study	
ID	CS-02
Objective	Verify that the system starts the monitorization mechanism in every process
Preconditions	The system must be executing
Sequence	Every process creates a thread to monitor the execution of the program
Postconditions	The monitorization thread is executing
Result	Verified
Trazability	FR-02

Table 5.3: Case study CS-02

Case study	
ID	CS-03
Objective	Verify that the system creates a binary tree shaped topology by recreating the same global tree in every process of the system and a local tree which would be equal for each node of the cluster in every process of the system
Preconditions	The monitorization mechanism must be running
Sequence	Every process creates two trees
Postconditions	The pair of trees is created in every process
Result	Verified
Trazability	FR-05, FR-09, NFR-05, NFR-06

Table 5.4: Case study CS-03

Case study	
ID	CS-04
Objective	Verify that the pair of binary trees created by each process is destroyed upon communication
Preconditions	The pair of binary trees must be created
Sequence	The pair of trees is deleted freeing their resources
Postconditions	The pair of trees is destroyed and their resources freed
Result	Verified
Trazability	FR-05, NFR-06 and NFR-07

Table 5.5: Case study CS-04

Case study	
ID	CS-05
Objective	Verify the communication among nodes works properly
Preconditions	The communication mechanism is started
Sequence	The data flows through the overlay
Postconditions	The data is sent among the different nodes in the overlay
Result	Verified
Trazability	FR-07, FR-08, NFR-02, NFR-03 and NFR-05

Table 5.6: Case study CS-05

Case study	
ID	CS-06
Objective	Verify that the data is aggregated by the node whose ID is 0
Preconditions	The communication mechanism is started
Sequence	The data flowing through the overlay is aggregated by the process whose ID is 0
Postconditions	The data is correctly aggregated by the node 0
Result	Verified
Trazability	FR-08, NFR-02 and NFR-03

Table 5.7: Case study CS-06

Case study	
ID	CS-07
Objective	Verify that the non relevant data is filtered by the processes and not send through the overlay
Preconditions	The communication mechanism is started
Sequence	The monitorization data is only sent to the parent process in the overlay if it is relevant, that is, if the new data is at least 5% higher or lower than the average of all the monitorization data read
Postconditions	The data which does not met the filtering condition is not sent through the network
Result	Verified
Trazability	FR-03

Table 5.8: Case study CS-07

Case study	
ID	CS-08
Objective	Verify the code is written in C
Preconditions	None
Sequence	Open the source code of the system and check it is written in C language
Postconditions	The code is written in C
Result	Verified
Trazability	NFR-01

Table 5.9: Case study CS-08

Case study	
ID	CS-09
Objective	Verify the code takes advantage of the EVpath library functionalities
Preconditions	None
Sequence	Open the source code of the system and check it is written using the EVpath header and functionalities
Postconditions	The code uses EVpath functions and data structures
Result	Verified
Trazability	NFR-02

Table 5.10: Case study CS-09

Case study	
ID	CS-10
Objective	Verify the code takes advantage of the MPI functionalities
Preconditions	None
Sequence	Open the source code of the system and check it is written using the MPI header and functionalities
Postconditions	The code uses MPI functions and data structures
Result	Verified
Trazability	NFR-03

Table 5.11: Case study CS-10

Case study	
ID	CS-11
Objective	Verify the monitorization data is updated on every iteration of the main loop
Preconditions	None
Sequence	Check how the value of the monitorization data changes periodically
Postconditions	The data value is updated on every iteration
Result	Verified
Trazability	FR-06

Table 5.12: Case study CS-11

Case study	
ID	CS-12
Objective	Verify the monitorization data is read every second on each process of the system
Preconditions	None
Sequence	Check how the value of the monitorization data is read every second
Postconditions	The data value is read on every second
Result	Verified
Trazability	FR-07

Table 5.13: Case study CS-12

Case study	
ID	CS-13
Objective	Verify the monitorization mechanism exits when the system finishes
Preconditions	None
Sequence	Check how all the execution threads launched by the monitorization mechanism finish upon termination of the system
Postconditions	The monitorization mechanism exits on finalization
Result	Verified
Trazability	FR-04

Table 5.14: Case study CS-13

Case study	
ID	CS-14
Objective	Verify the system runs CPU intensive applications, communication intensive applications and I/O intensive applications
Preconditions	None
Sequence	Run the system with different size of matrices and different number of processes
Postconditions	The system runs these kind of applications
Result	Verified
Trazability	NFR-08, NFR-09 and NFR-10

Table 5.15: Case study CS-14

Case study	
ID	CS-15
Objective	Verify that the relevant monitorization data is shown on the screen
Preconditions	The system must be running
Sequence	The application is running and the node whose ID is 0 is showing on the screen the non filtered data aggregated through all the overlay
Postconditions	The system has shown the data on the screen
Result	Verified
Trazability	FR-01

Table 5.16: Case study CS-15

5.3 Traceability matrix

	FR-01	FR-02	FR-03	FR-04	FR-05	FR-06	FR-07	FR-08	FR-09	NFR-01	NFR-02	NFR-03	NFR-04	NFR-05	NFR-06	NFR-07	NFR-08	NFR-09	NFR-10
CS-01												X	X				X	X	X
CS-02		X																	
CS-03					X			X						X	X				
CS-04					X				X						X	X			
CS-05							X	X			X	X		X					
CS-06								X			X	X							
CS-07			X																
CS-08										X									
CS-09											X								
CS-10												X							
CS-11						X													
CS-12							X												
CS-13				X															
CS-14																	X	X	X
CS-15	X																		

Table 5.17: Traceability matrix

5.4 Performance tests

In this section, the different tests performed to test the performance of the developed system are presented. First, a simple usage test in which the different high level parts of the system are checked is shown. Then a test performed on the throughput of the communication system is presented. Finally, a test in which the filtering functionality is tested is explained.

5.4.1 Usage test

In this section, the three following questions are tested:

- Does the system compile?
- Does the system execute?
- Does the system send data through the network overlay?

The table 5.18 shows the results of the three tests. As we can see, all the three tests are passed as shown by the green color of the cells.

Use test	
Compiles?	
Executes?	
Sends data?	

Table 5.18: Results of the usage tests

5.4.2 Throughput tests

For testing the throughput, the system has been executed in three different configurations, in which the time for the sending nodes to send data varies among *0.01*, *0.5* and *1* seconds. Along these three configurations, the system has been executed with a different number of processes, being them *1*, *2*, *4*, *8*, *16*, *24*. Finally, these tests have been executed for a configuration in which the system runs entirely in the same machine, and another one in which the system runs in four different nodes of the Tucán cluster. Also, in order to get all the throughput, the filtering mechanism has been disabled for performing these tests.

Additionally, for comparison, an alternative network architecture has been used in order to compare the results with our proposed of tree-shaped network architecture. This comparative configuration is the same as the one stated in the Introduction of this document, shown in figure 1.1, in which all the processes of the application send the data directly to the 0 ID process.

Intra node test

For the test in which the system is executed in the same machine, the table 5.19 shows the results for the different configurations, that is, the intra node execution results. These results show the number of communications that arrive to the process whose ID is 0, that is, the process which aggregates all the communication and shows them in the screen.

	1 node	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**24 nodes**
0.01 secs	1092	2278	5260	9443	19151	35200
0.5 secs	430	674	840	2392	4010	6322
1 sec	216	282	516	976	2064	2433

Table 5.19: Intra node throughput execution tests

The results for this test can be easily analyzed in the figure 5.1. This figure shows the number of communications aggregated by the 0 ID process in a period of 2 minutes approximately for the different configurations. Here where we can appreciate that with more processes executing in the system, and with more frequency of sending data, the number of communications aggregated grows. This makes sense, as with more processes to monitor, more data is going to be generated, and with a lower period of sending this data, the higher the amount of data to be received over time.

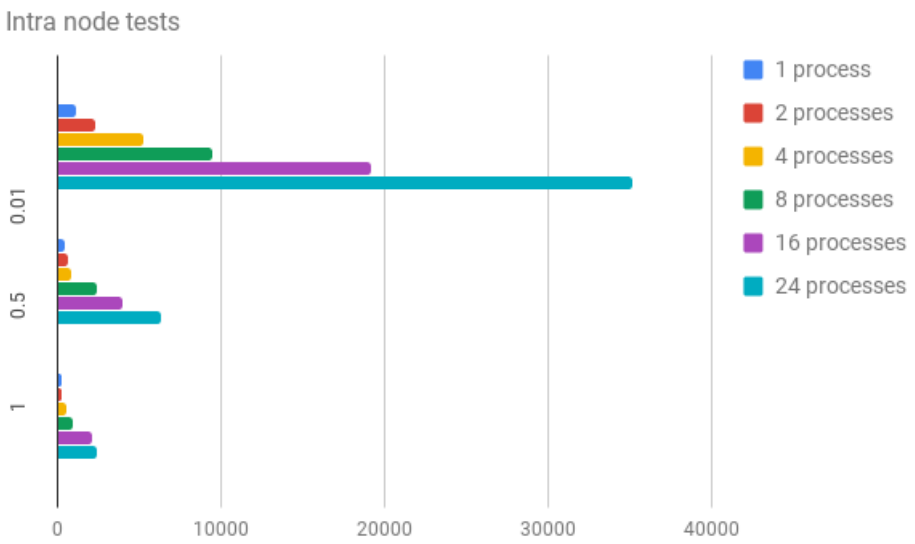

Figure 5.1: Intra node throughput execution tests diagram

For the test in which the system is executed in different nodes, the table 5.20 shows the results for the different configurations, that is, the intra node execution results. These results show the number of communications that arrive to the process whose ID is 0, that is, the process which aggregates all the communication and shows them in the screen.

	1 node	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**24 nodes**
0.01 secs	1093	2378	5730	8576	16464	25728
0.5 secs	431	718	896	2152	3280	4704
1 sec	215	236	524	904	1624	2712

Table 5.20: Inter node throughput execution tests

The results for this test can be easily analyzed in the figure 5.2. This figure shows similar results to 5.1, as it draws the same conclusions, but in a lower scale for higher amount of processes being the maximum number of communications in this configuration *25728*, while for the previous configuration it was *35200*. For lower amount of nodes, the results are far more similar.

These differences could be explained by the fact that the communication inter nodes in the cluster is far more costly than inside the same node because, as explained in the section 1.2.2 of the Introduction of this document, the different machines of the cluster are connected with switches whose speed varies, so that, the monitorization mechanism spends more time in communications in this configuration than in the previous one.

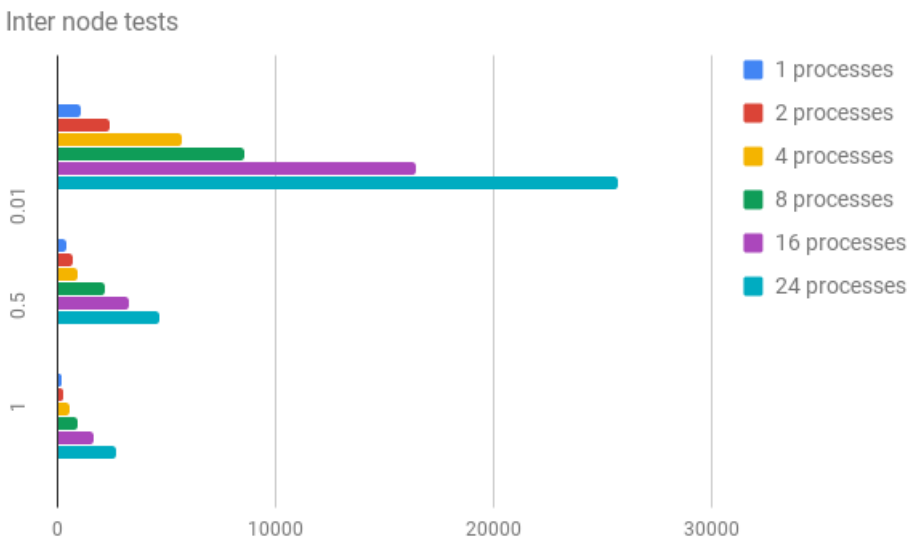

Figure 5.2: Inter node throughput execution tests diagram

As comparison the table 5.21 shows the results of the execution of the tests in the intra node configuration for the alternative architechture.

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	24 nodes
0.01 secs	13075	13312	35429	84873	157955	235685
0.5 secs	262	264	705	1680	4109	6056
1 sec	131	133	351	833	2490	303

Table 5.21: Inter node throughput execution tests

The results for this test can be easily analyzed in the figure 5.3. These results show how in this architecture the throughput of messages depends much more on the frequency of sending messages, as there is a much more significative difference on throughput between the configuration in which the sending frequency is of *0.01* seconds and the other two configurations, than this difference in the tests 5.1.

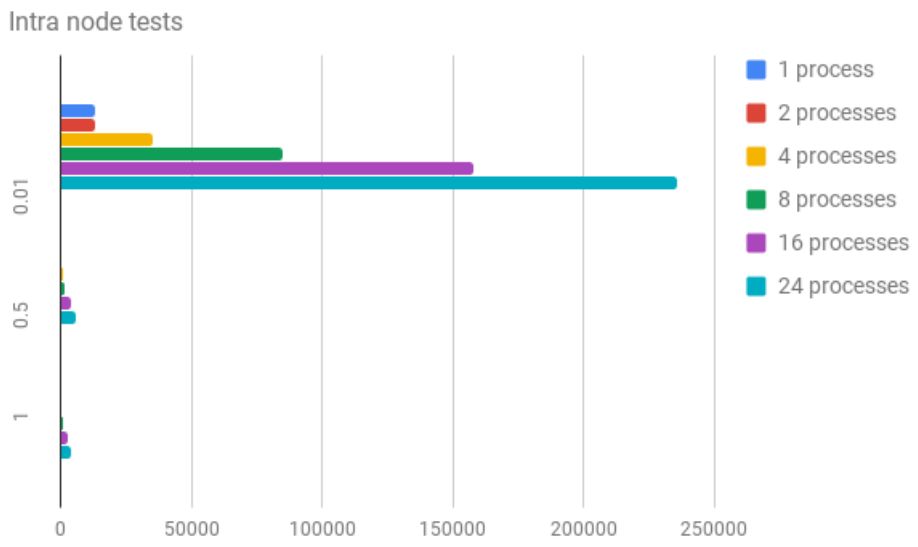

Figure 5.3: Intra node throughput execution tests diagram in alternative architecture

The table 5.22 shows the results of the execution of the tests in the inter node configuration for the alternative architecture.

	1 node	**2 nodes**	**4 nodes**	**8 nodes**	**16 nodes**	**24 nodes**
0.01 secs	13044	13368	50477	77479	177718	248791
0.5 secs	262	278	759	1575	3390	5106
1 sec	131	142	378	791	1725	2714

Table 5.22: Inter node throughput execution tests

The results for this test can be easily analyzed in the figure 5.4. This shows similar results than in figure 5.3 being the throughput measured really similar, being the ones performed in this tests slightly smaller than the other one. This, as in the case of the proposed architecture can be explained by the cost of the communication among processes which reside in different nodes of the cluster.

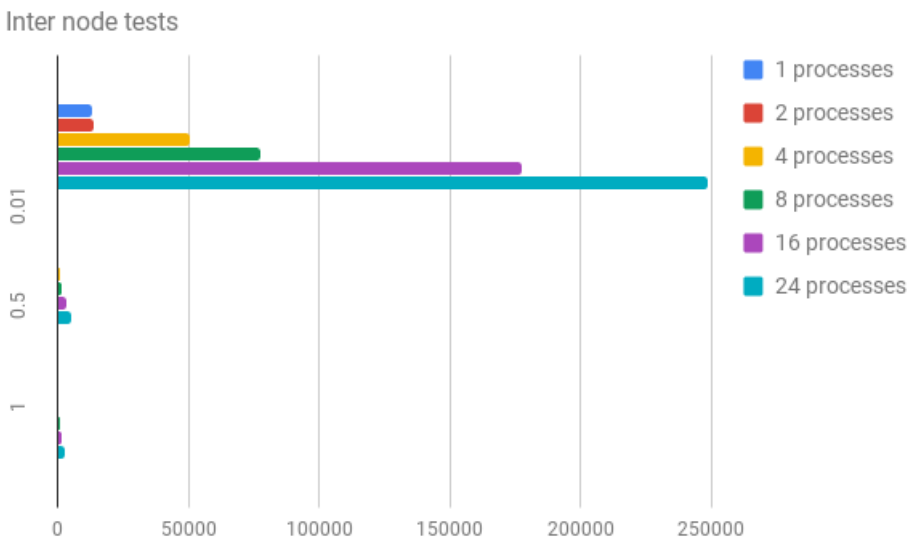

Figure 5.4: Inter node throughput execution tests diagram in alternative architecture

These differences in the measured throughput between the two architectures can be related to the scalability of the systems. While in the proposed solution, a process which represents a leaf in its Local Tree sends a message in the same way as in the alternative architecture any node would do, this message sent by the leaf process have to travel through more nodes of the overlay, which makes the system use a higher amount of communications. This would represent a trade off between scalability and throughput.

Filtering test

For testing the functionality of the filtering mechanism, the following test is presented. The alternative network architecture employed for comparison in the throughput tests has also been employed.

To test the difference between the two architectures, they have been executed with the same filtering mechanism configured to filter the messages to send differently. For this, six different filtering thresholds have been tested. These filtering thresholds are *0.01*, *0.02*, *0.05*, *0.1*, *0.25* and *0.5*. These thresholds mean that if a message is to be sent from one process to another, if it is not at least, for example, 2% higher or lower (0.02 threshold) than the average of the previously sent messages, it is discarded, if not it is sent.

The table 5.23 shows the number of communications aggregated by the zero ID process for the different filtering thresholds execution in the two different architectures.

	0.01	**0.02**	**0.05**	**0.1**	**0.25**	**0.5**
Tree based monitorization	1495	1442	636	359	179	58
Centralized Monitorization	1973	1544	625	298	178	124

Table 5.23: Filtering test results

The results for these tests can be easily analyzed in the figure 5.5, where we can see how for higher filtering thresholds, the two architectures behave similarly, aggregating similar number of communications, while for lower thresholds the proposed architecture receives a smaller amount of communications. This means that the proposed architecture can be far more precise than the alternative one, as with a filtering threshold which would only filter very redundant communications we can reduce the number of these communications considerably.

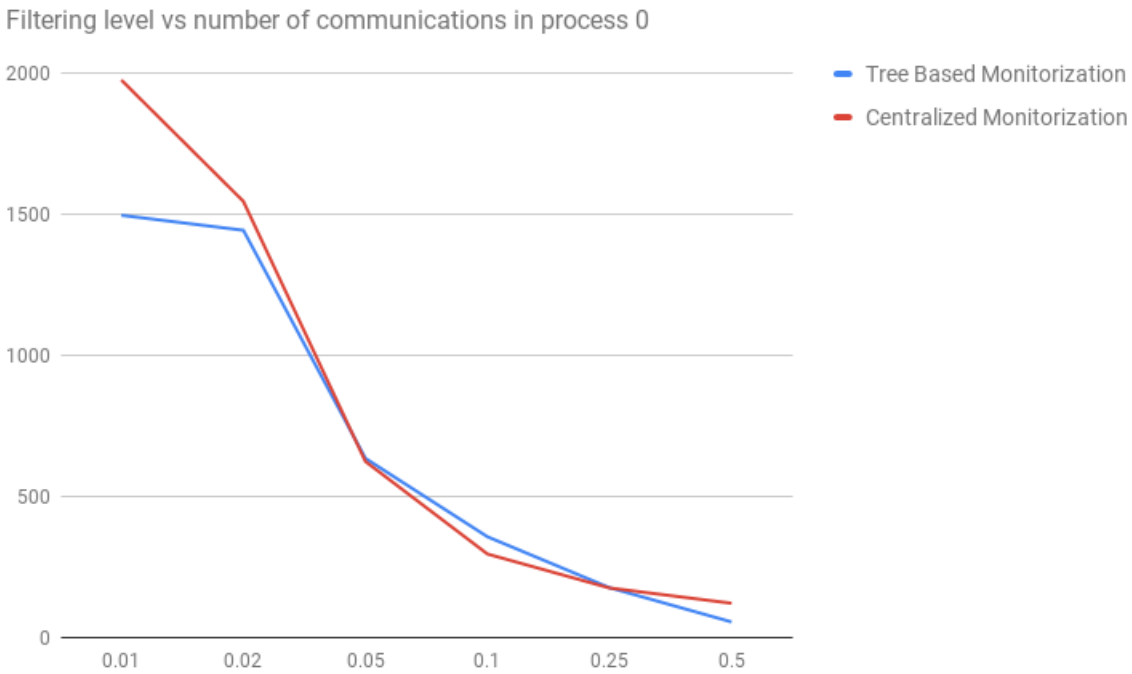

Figure 5.5: Filtering test results diagram

Chapter 6

Conclusions and future work

In this chapter the conclusions drawn in the development of the project are presented alongside with the future work that can be taken.

6.1 General conclusions

The main objective of this work consists on building a scalable communication mechanism for MPI applications.

In order to achieve this main contributions have been proposed:

- A mechanism to create a binary tree shaped network topology for MPI applications which is aware of the hardware configuration which has more or less nodes depending on the number processes to run and the number of nodes of a cluster to use.
- A communication mechanism for these kind of applications which takes advantage of the network topology configured to communicate the different processes applying a filter to the data to be transmitted. This filter makes the mechanism discard all the irrelevant data and avoids sending it.

These two contributions have reduce significantly the problem stated in 1.3 in which the problematic of aggregating all the data sent by the different processes by one them, without taking into account the network bottleneck that is created.

With the proposed solutions, the processes send to up to two other processes maximum, and receives from up to four of them. This, added to the filter mechanism which avoids a significant amount of communications makes the network bottleneck to not being a problem anymore. The filter mechanism also comes with a downside. As this communication mechanism has been applied to a monitorization system, the filtering of data makes the system to avoid some communications.

To sum up, the two objectives stated in 1.3 were :

- Implement a scalable data communication mechanism with low overhead
- The mechanism must be conscious of the topology of the machine

The first objective was reached by the implementation of the binary tree shaped communication topology, which avoids the network bottleneck illustrated in the figure 1.1, as it reduces the cost of aggregating the data in one node.

The second objective was reached by the usage of a Balanced Binary Search Tree data structure as a base to create the binary tree shaped communication topology. With this decision, the information of processes to run can be introduced in different binary trees in order for the different processes to know the topology of all the system and perform the communication.

This work has required knowledge in Distributed Systems, C programming, data Structures and algorithms and software engineering. Developing this project has supposed a challenge in which the acquisition of knowledge, and also the re freshening and expansion of already acquired knowledge, has been notable.

The results obtained have been satisfactory, with some future work that, for timing reasons, has not been able to be implemented, which is stated in 6.2.

6.2 Future work

In this section, the future work to be added in order to improve the developed system functionality is presented:

- Improve the actualization of the monitorization data by, instead of keeping track of the time to execute an iteration of the main loop of the application by calculating a weighted average of the calculated values, implement a mechanism to store the last N read values in a circular buffer data structure. This would improve the precision of the data used to monitor the performance of the application.

- Improve memory management. Despite the system developed memory management being fairly efficient, the usage of the third-party library EVpath has presented some issues with memory management, as the implementation of this library present some memory leaks, whose fix is not in the scope of this project. Improving this would present a challenging scenario whose easier fix could be to drop EVpath library usage for another third-party library with better memory management, or even an own developed solution.

Appendix A

User Manual

The steps reproduced to use the proposed software by a user are the following:

1. Navigate to the folder where the project is
2. Compile the software:

```
$ make
```

3. Execute the software using the following structure:

```
$ mpiexec -np <number of processes> <<optional>>-f <path to rankfile>
<<optional>> ./jacobi <size of matrix> <number of iterations>
<convergence criteria> <CPU complexity> <communications complexity>
<I/O complexity>
```

An example of execution with four processes would be:

```
$ mpiexec -np 4 -f rankfile ./jacobi 500 1000 0.00001 1 1 0
```

Whose results are:

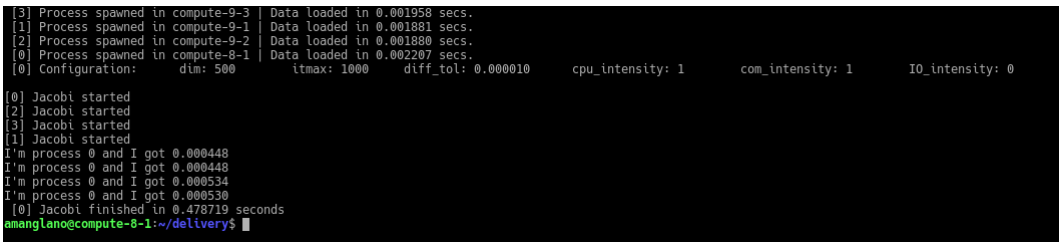

Figure A.1: System execution example

Appendix B

Developer Manual

In order to use the proposed software the required dependencies must be installed.

As stated, the project itself utilizes directly, the MPICH implementation of MPI and the EVpath library. MPICH does not have any special dependency, but EVpath have some whose installation will be detailed below.

For simplicity, the same folder is used to install all the required dependencies. This folder would be referred as $ENV and its path would be:

```
/home/<user folder>/usr
```

In this folder, by installing the libraries, three standard subfolders will be created:

- bin: Where the binaries of the libraries are stored
- lib: Where the subroutines (.a files for the static libraries and .so for the dynamic ones) generated are stored
- include: Where the library headers for the C programming language are stored

So, for adding these folders to the environment path, the file .profile in the user's home directory must be modified by adding the following lines:

```
export PATH=$PATH:~$ENV
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~$ENV/lib
export LIBRARY_PATH=$LIBRARY_PATH:~$ENV/lib
export C_INCLUDE_PATH=$C_INCLUDE_PATH:~$ENV/include
export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:~$ENV/include
```

B.1 MPICH

The following steps must be performed to install MPICH:

1. Download MPICH from here.
2. Uncompress it:

```
$ tar xf mpich-XXXX.tar.gz
```

3. Navigate to the MPICH folder:

```
$ cd mpich-XXXX
```

4. Configure the installation choosing the desired installation folder:

```
$ ./configure --prefix=$ENV
```

5. Build MPICH:

```
$ make
```

6. Install MPICH:

```
$ make install
```

B.2 EVpath

EVpath has some dependencies from packages of the Georgia Institute of Technology. The packages are the following:

- dill
- cercs_env
- atl
- ffs

The ffs library has itself some other dependencies:

- GNU Bison
- flex

In order to install all these environment, it is recommended to install first the ffs dependencies and then the rest of the packages from the Georgia Institute of technology. The steps for building the latter are the same for all the packages, including EVpath, so it will only be explained once, while the steps for installing Bison and Flex will be explained separately:

B.2.1 GNU Bison

The following steps must be performed to install Bison:

1. Download GNU Bison from here.
2. Uncompress it:

```
$ tar xf bison-XXXXX.tar.xz
```

3. Navigate to the Bison folder:

```
$ cd bison-XXXXX
```

4. Configure the installation choosing the desired installation folder:

```
$ ./configure --prefix=$ENV
```

5. Build Bison:

```
$ make
```

6. Install Bison:

```
$ make install
```

B.2.2 Flex

The following steps must be performed to install Flex:

1. Download Flex from here.
2. Uncompress it:

```
$ tar xf flex-XXXXX.tar.gz
```

3. Navigate to the Flex folder:

```
$ cd flex-XXXXX
```

4. Execute the autogen script:

```
$ ./autogen.sh
```

5. Configure the installation choosing the desired installation folder:

```
$ ./configure --prefix=$ENV
```

6. Build Flex:

```
$ make
```

7. Install Flex:

```
$ make install
```

B.2.3 Georgia Tech Libraries

Now that Bison and Flex are installed properly, the installation for dill, cercs_env, atl, ffs and EVpath is explained.

1. Download the packages:
   (a) dill
   (b) cercs_env
   (c) atl
   (d) ffs
   (e) EVpath
2. Uncompress the package:

```
$ tar xf vXXXXX.tar.gz
```

3. Navigate to the folder:

```
$ cd vXXXXX
```

4. Create a building folder:

```
$ mkdir build
```

5. Navigate to the building folder:

```
$ cd build
```

6. Configure the installation choosing the desired installation folder:

```
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=$ENV ..
```

7. Build and install:

```
$ make all install
```

Applying these steps for the five packages, should install EVpath and all its dependencies.

Bibliography

[1] e. a. Keren, Bergman, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency, Information Processing Techniques Office*, 2008.

[2] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," *International Meeting on High Performance Computing for Computational Science*, vol. 6449, 2011.

[3] M. Gerndt, E. Fürlinger, and E. Kereku, "Periscope: Advanced techniques for performance analysis," *Current and Future Issues of High-End Computing*, vol. 33, 2206.

[4] D. Schmidl, C. Terboven, D. an Mey, and M. S. Müller, "Suitability of performance tools for openmp task-parallel programs," 2013.

[5] Intel, "Intel® vtune™ amplifier xe," 2013.

[6] M. Geimer, F. Wolf, B. J.N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "Suitability of performance tools for openmp task-parallel programs," *Concurrency and Computation: Practice and Experience - Scalable Tools for High-End Computing*, vol. 22, pp. 702–719, 2010.

[7] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller, and W. Nagel, "The vampir performance analysis tool-set," pp. 139–155, 01 2008.

[8] W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," vol. 12, 05 1996.

[9] S. Shende and A. Malony, "The tau parallel performance system.," vol. 20, pp. 287–311, 01 2006.

[10] D. D'arquitectura De Computadors, V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," vol. 44, 03 1995.

[11] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using papi for hardware performance monitoring on linux systems," 08 2009.

[12] B. J. Nelson, "Remote procedure call," 1981. AAI8204168.

[13] J. Waldo, "Remote procedure calls and java remote method invocation," *IEEE Concurrency*, vol. 6, pp. 5–7, Jul 1998.

[14] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad, "Efficient implementations of java remote method invocation (rmi)," pp. 2–2, 1998.

[15] S. Vinoski, "Corba: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications Magazine*, vol. 35, pp. 46–55, Feb 1997.

[16] w3.org, "Web services glosary," 2004.

[17] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications, Berlin: Springer, 2004.

[18] F. Sunday Emmanuel, "On some iterative methods for solving systems of linear equations," vol. 1, pp. 21–28, 2015.

[19] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," vol. 14, pp. 328–339, 01 1987.

[20] E. Ministerio de Empleo y Seguridad Social, "Ley orgánica 15/1999, de 13 de diciembre, de protección de datos de carácter personal.," 1999.

[21] mpich.org, "Mpich license."

[22] O. S. Initiative, "The 3-clause bsd license."

[23] E. Ministerio de Empleo y Seguridad Social, "Seguridad social:trabajadores," 2017.

[24] E. Ministerio de Empleo y Seguridad Social, "Xviii convenio colectivo nacional de empresas de ingenierÍa y oficinas de estudios tÉcnicos," 2017.

[25] M. Corporation, "Mpl 2.0 faq," 2017.

[26] M. Corporation, "Mozilla public license version 2.0," 2012.